目　录

第一章　YFARM9-EDU-I 主板说明

一、系统概述

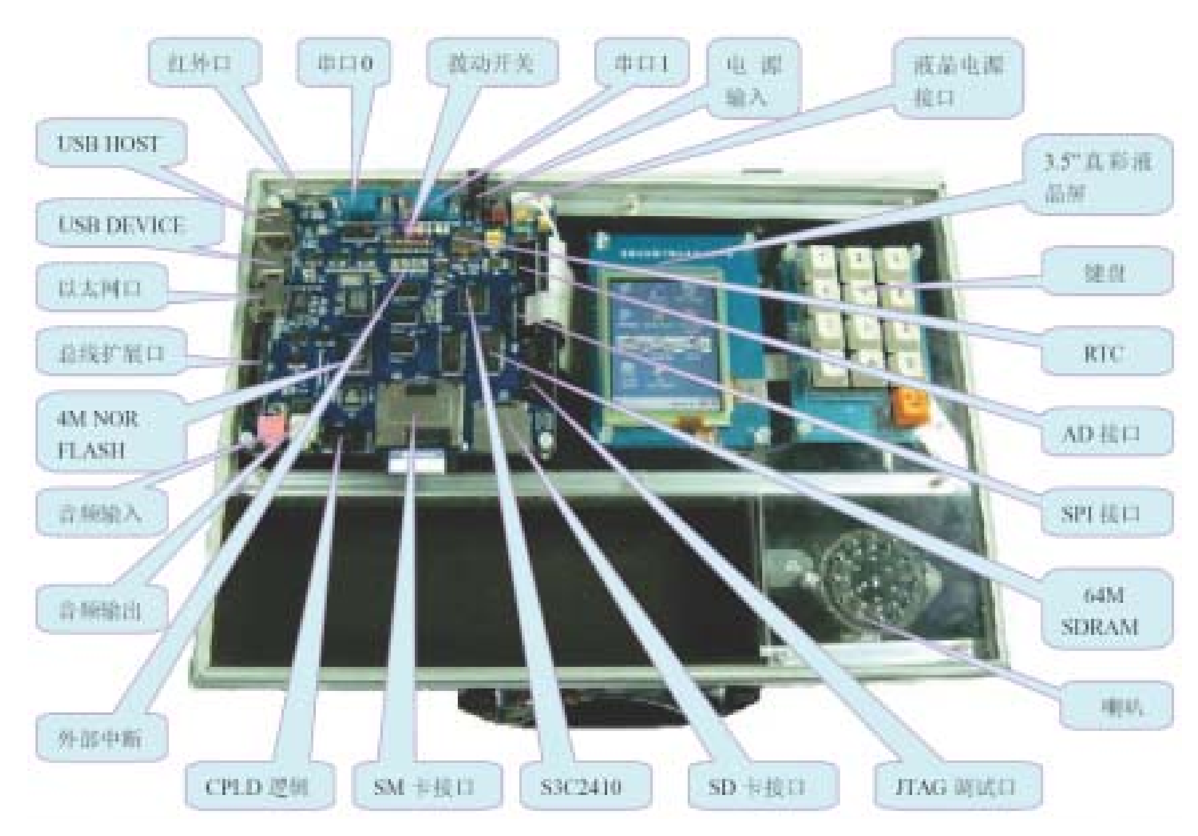


图 1-1：YFARM9-EDU-I 主板

YFARM9-EDU-I 是一款基于 Samsung 高性能 CPU－S3C2410（ARM920T）的实验平台，接口丰富，具有一下特性：

（1）　CPU ：S3C2410（16-/32-bit ARM920T 内核）
（2）　系统时钟：使用外部 12MHz 晶体由 CPU 内部 PLL 备频至 200MHz+
（3）　BOOT ROM：
　　Intel StrataFlash ：4Mbyte）E28F320J3
　或
　　Smart Media Card
（4）　SDRAM ：64Mbyte（32Mbyte×2）
（5）　3.5 寸 标准 PDA 用 240×320 64K 色真彩 TFT LCD 和触摸屏控制器
（6）　3 信道 UART（已包含 IRDA 红外线数据通讯口）
（7）　2 个 USB 主机控制器（其中一个可配置为 USB 设备控制器）
（8）　SD 卡 / MMC 卡主机控制器
（9）　Smart Media Card 控制器
（10）　Embedded-ICE 调试接口（标准 20pin JTAG）
（11）　RTC 实时时钟（具备后备锂电池）
（12）　IIC 总线接口（驱动 AT24C04-SC27）
（13）　ADC 模数转换接口
（14）　SPI 接口（亦可定义为通用 IO 口扩展矩阵键盘）
（15）　IIS 数字音频输入/输出接口

（16） 多功能总线扩展接口（含 EINT 外部中断接口）
（17） IrDA 红外线收发器
（18） 4 个板上轻触键
（19） 10M 以太网接口
（20） 5 只发光二极管指示灯

YFARM9-EDU-I 向您充分展示了如何基于 S3C2410 进行系统级的硬件、软件设计，并且可使您在该平台的基础上迅速地开展您的自己的产品设计。

图 1-2 所示是 YFARM9-EDU-I 主系统的功能框图。

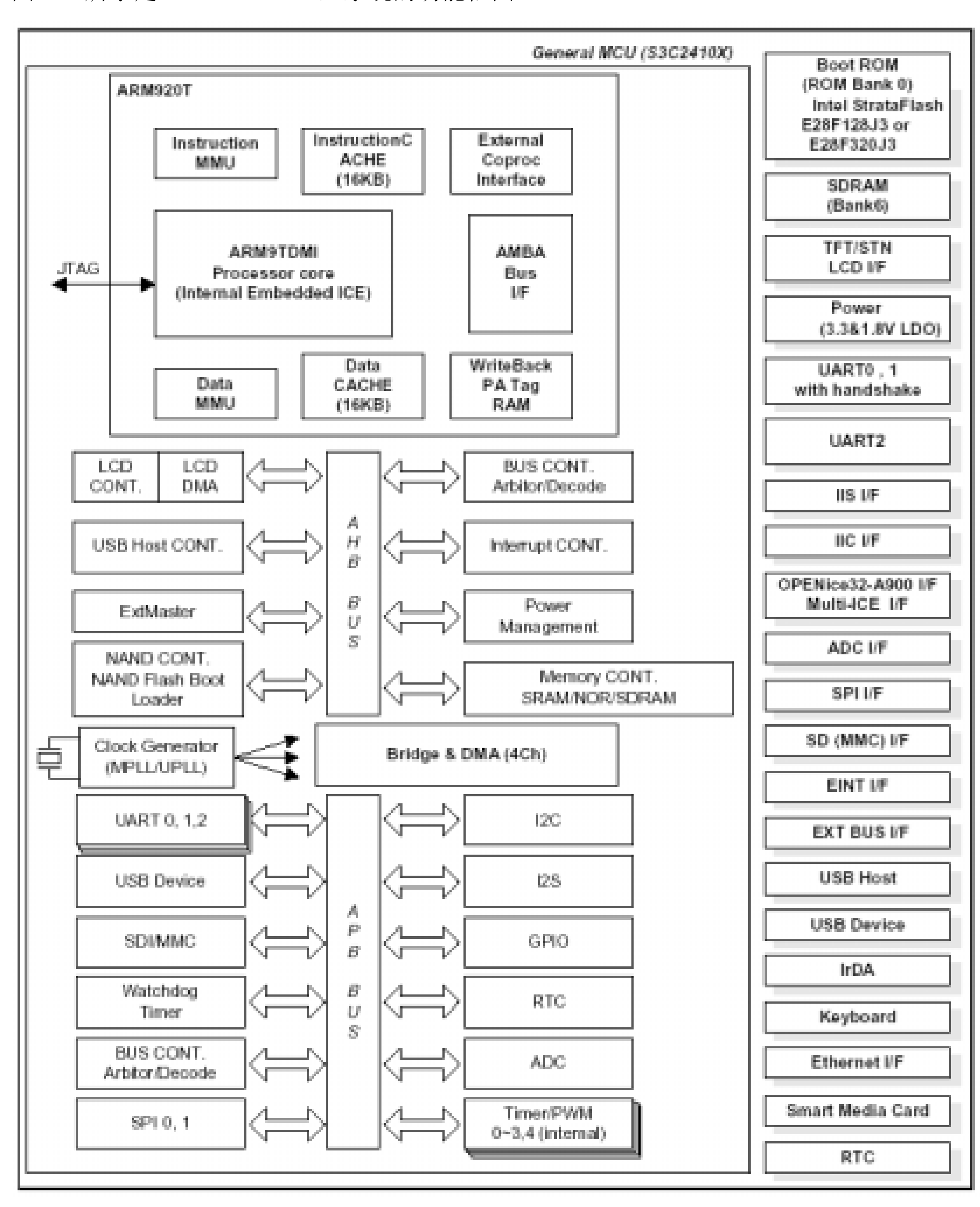


图 1-2：YFARM9-EDU-I 主系统功能框图

二、YFARM9-EDU-I 主板电路说明

图 1-3 所示为 YFARM9-EDU-I 主板部件示意图

系统供电

YFARM9-EDU-I 主板由外部提开关电源提供 5V 直流电源，经由 1.5A 自恢复保险丝并通过开发板内 LDO 分别稳压至 1 路 3.3V 和 2 路 1.8V。同时保险丝后端的 5V 直流电源还向 USB 主机接口和 LCD 接口提供电源。而 RTC（实时时钟）考虑到需要在开发板断电后仍能继续工作，故采用独立的锂电池供电。

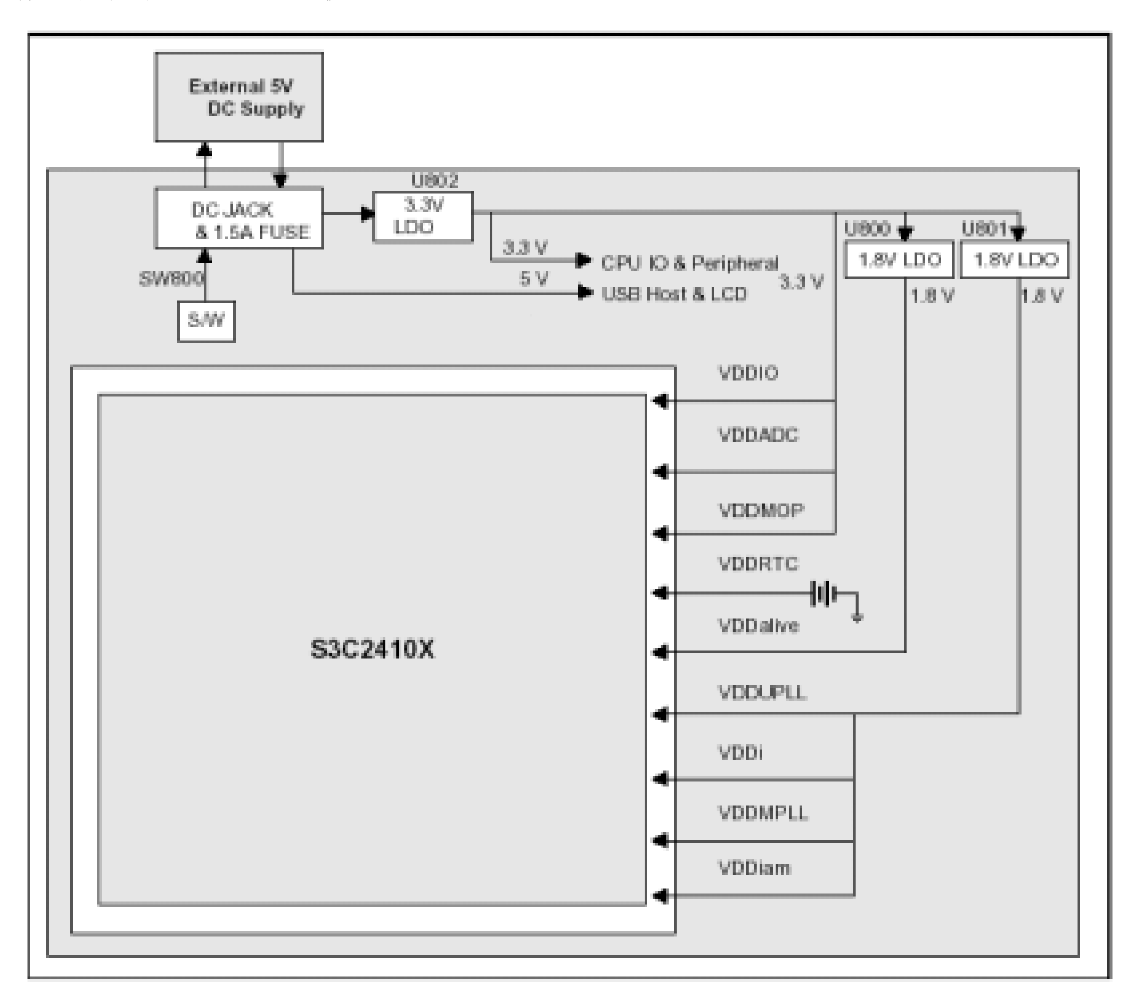


图 1-3：YFARM9-EDU-I 主板供电示意图

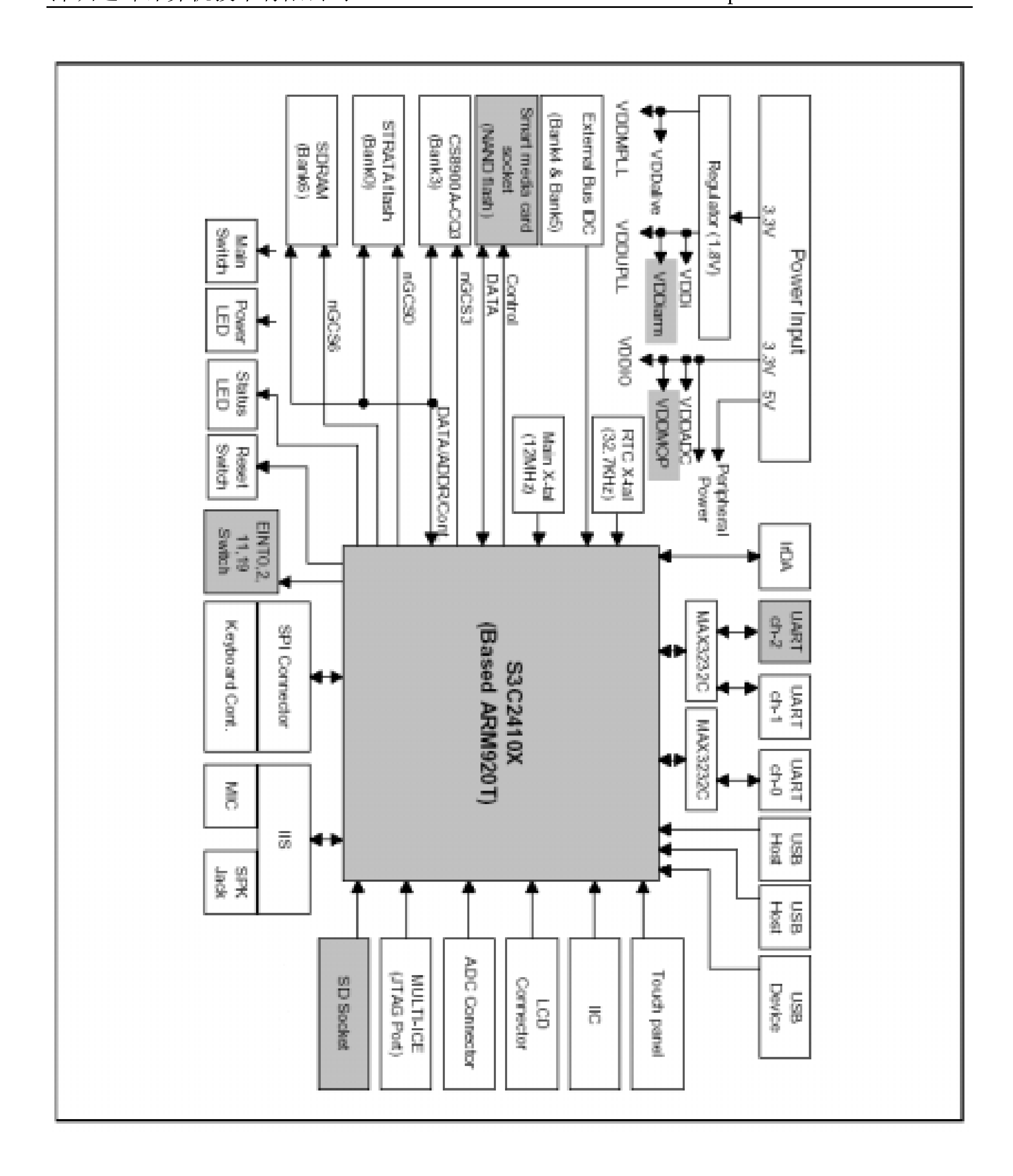


图 1-4：YFARM9-EDU-I 主板部件示意图

三、YFARM9-EDU-I 主板系统设置

主时钟源

S3C2410 可以使用无源晶体（X100）或有源晶体（由 S100 的 PIN1 引入）作为系统 PLL 和 USB PLL 的基准输入时钟源总线时钟，并经内部 PLL 产生 CPU 的工作时钟和 USB 的。所有这些设置均可通过对 CPU 管脚 OM[3:2]来选择具体的时钟模式。

表 1-1

S100-1	S100-5	S100-4	OM3	OM2	功能描述
ON	ON	ON	0	0	MPLL：XTAL，UPLL：XTAL
OFF	ON	OFF	0	1	MPLL：XTAL UPLL：EXTCLK
OFF	OFF	ON	1	0	MPLL：EXTCLK，UPLL：XTAL
OFF	OFF	OFF	1	1	MPLL：EXTCLK，UPLL：EXTCLK

实时时钟 RTC

实时时钟 RTC 采用外部 32.768KHz 无源晶体（X101）作为 RTC 时钟。

注意：

1、 尽管系统时钟 MPLL 在 CPU 复位之后就开始工作，但是 MPLL 的输出直到软件向 MPLLCON 寄存器写入有效值之后才真正作为系统的主时钟。在此之前，系统时钟直接从外部晶体频率或者 EXTCLK 获得。因此尽管用户想仍然保留原来的时钟设置，也应该在 CPU 复位之后重新将 MPLLCON 原来的值写入 MPLLCON 寄存器（例如在执行 SoftReset 之后）。

2、 当 OM[1:0]为 11 时，OM[3:2]作为 CPU 测试模式的选择。

选择 BootRom 选择

S3C2410 可配置为从以下的 Flash ROM 中启动，并且是业界第 1 颗支持直接从 NAND Flash 启动的处理器（注：NAND Flash 是针对其 Flash 的结构而言的，它可以是芯片－IC 的形式，也可以是存储卡－Smart Media Card 的形式）。

表 1-2

S100-3	S100-2	OM1	OM0	功能描述
ON	ON	0	0	由 NAND Flash Boot
ON	OFF	0	1	由 16bit NOR Flash Boot
OFF	ON	1	0	由 32bit NOR Flash Boot
OFF	OFF	1	1	CPU 进入测试模式

选择 NAND Flash 类型

NAND Flash 是按页（Page）寻址的 Flash 类型，根据 Flash 的容量不同，分别有 3 步寻址模式和 4 步寻址模式之分。这方面的相关内容可参阅 NAND Flash 的 Datasheet。

表 1-3

S100-6	NCON	功能描述	备注
ON	0	3步寻址模式	支持≤32Mbyte 的 NAND Flash
OFF	1	4 步寻址模式	支持≥64Mbyte 的 NAND Flash

S3C2410 从工程样片到目前的正式投产经历了几个版本。在这几个版本中最明显的差异在于对 NAND Flash nR/B（nBUSY）管脚的处理。因此 YFARM9-EDU-I 主板可灵活配置 nR/B（nBUSY）的连接方式。

表 1-4

S100-9	S100-10	NAND Flash’s nR/B（nBUSY）描述
ON	OFF	连接至 CPU 的 RnB 管脚
OFF	ON	连接至 CPU 的 nWAIT 管脚

设置 NOR Flash 的写保护

YFARM9-EDU-I 主板内置了 1 颗 Intel 的 4Mbyte StrataFlash，型号为 E28F320J3。在 YFARM9-EDU-I 主板出厂时，在该芯片内部已经预装了 USB Downloader 的固件代码（FirmWare)。用户也可以在该 Flash 中装载自己的代码。为了防止该 Flash 中的内容被用户错误的程序意外修改或擦除，在 YFARM9-EDU-I 主板中特地设置了 Flash 的写保护开关。

表 1-5

S100－7	VPEN	功能描述
ON	0	禁止对 NOR Flash 进行写、擦除操作
OFF	1	使能对 NOR Flash 进行写、擦除操作

通用 IO 口

在 YFARM9-EDU-I 主板上分别使用了 4 个输出口和 4 个输入口扩展了 LED 指示灯和轻触键。

表 1-6

IO 埠号	IO 类型	功能描述
GPF[7:4]	输出	LED显示
GPF0，GPF2，GPG3和GPG11	输入	轻触键输入（EINT0，2，11和19）

四、YFARM9-EDU-I 主板外围扩展口说明

LCD&触摸屏接口

S3C2410 内置的 LCD 控制器支持 STN 和 TFT 的各种规格 LCD 屏，同时还具备了触摸屏控制器。为了方便用户自行扩展所需的 LCD 屏以及触摸屏，SamAemDvk9 将 CPU 的所有 LCD 控制信号连同触摸屏接口由统一的 50PIN 接插件 CON702 引出。CON702 的管脚定义如下：

表 1-7

管脚	CPU	STN	TFT	管脚	CPU	STN	TFT
1	-	VDD33	VDD33	2	-	VDD33	VDD33
3	-	VDD33	VDD33	4	-	GND	GND
5	-	nRESET	nRESET	6	VD0	VD0	B0
7	VD1	VD1	B1	8	VD2	VD2	B2
9	VD3	VD3	B3	10	VD4	VD4	B4
11	VD5	VD5	B5	12	VD6	VD6	B6
13	VD7	VD7	B7	14	VD8	VD8	G0
15	VD9	VD9	G1	16	VD10	VD10	G2
17	VD11	VD11	G3	18	GND	GND	GND
19	VD12	VD12	G4	20	VD13	VD13	G5
21	VD14	VD14	G6	22	VD15	VD15	G7
23	VD16	VD16	R0	24	VD17	VD17	R1
25	VD18	VD18	R2	26	VD19	VD19	R3
27	VD20	VD20	R4	28	VD21	VD21	R5
29	VD22	VD22	R6	30	VD23	VD23	R7
31	GND	GND	GND	32	LCD_PWR	LCD_PWR	LCD_PWR
33	LCDVF2	—	LCDVF2	34	LCDVF1	—	LCDVF1
35	LCDVF0	—	LCDVF0	36	VM	VM	VDEN
37	VFRAME	VFRAME	VSYNC	38	VLINE	VLINE	HSYNC
39	VCLK	VCLK	VCLK	40	LEND	—	LEND
41	nDISP	nDISP	nDISP	42	GND	GND	GND
43	XMON	XMON	XMON	44	nXPON	nXPON	nXPON
45	AIN7	AIN7	AIN7	46	GND	GND	GND
47	YMON	YMON	YMON	48	nYPON	nYPON	nYPON
49	AIN5	AIN6	AIN5	50	GND	GND	GND

注：表中 TFT 的管脚定义是按照 24BPP（24 位色）来表示的，当外部扩展 16BPP 或 18BPP 的 TFT LCD 时，按照 MSB 对齐的模式来连接（高有效位对齐）。

下图为 S3C2410 内部触摸屏控制器的功能示意图：

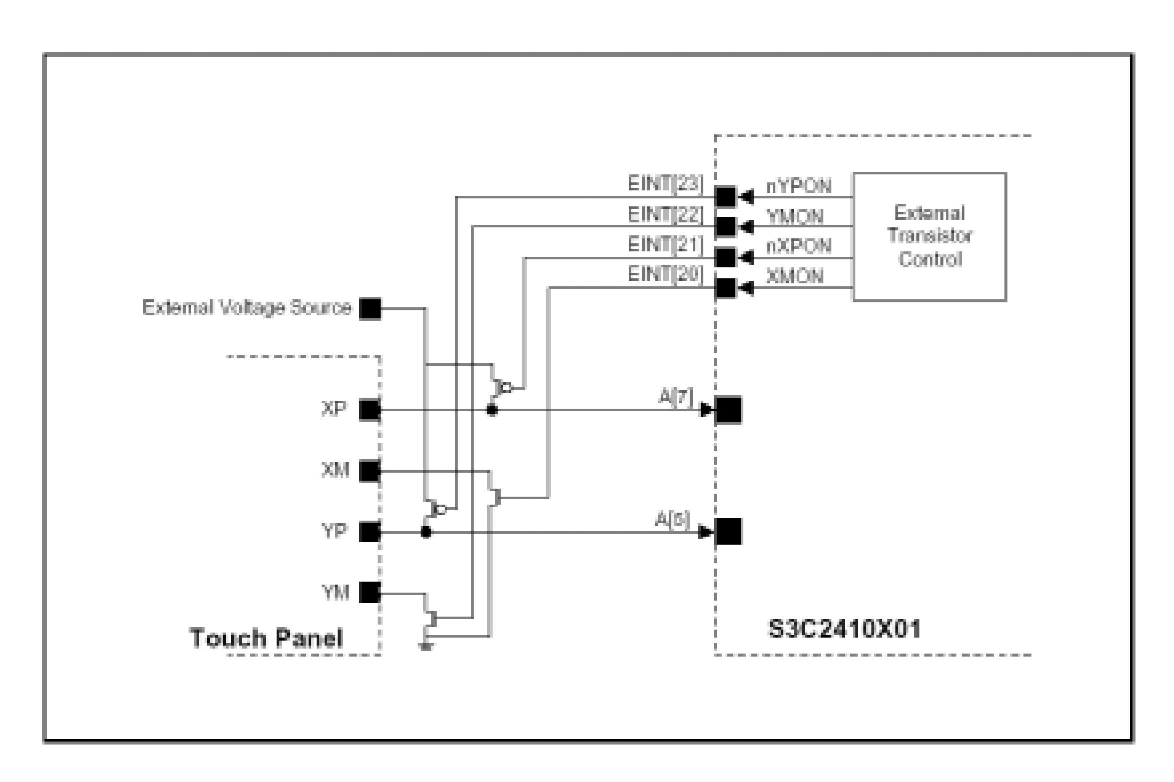


图 1-5：触摸屏控制器功能示意图

串口（UART）及红外数据通讯口（IRDA）

S3C2410 具有 3 信道的 UART，其中 UART2 可配置为红外数据通讯口（IRDA）。相关的硬件设置如下：(0=断开，C＝短接)

表 1-8

引脚功能	SEL600A		SEL600B		SEL600C		SEL600C		功能描述
	A-B	B-C	A-B	B-C	A-B	B-C	A-B	B-C	
UART 定义	0	C	C	0	0	C	C	0	CON600:UART0 CON601:UART1
	C	0	-	-	C	0	-	-	CON600:UART0 CON601:UART2
	-	-	0	C	-	-	0	C	CON600:Modem

当 UART2 定义为 IRDA 口时，

表 1-9

引脚功能	SEL600E		SEL600F		功能描述
	A-B	B-C	A-B	B-C	
UART2	0	C	0	C	UART2
定义	C	0	C	0	IRDA

IRDA 的原理图参见图 1-5

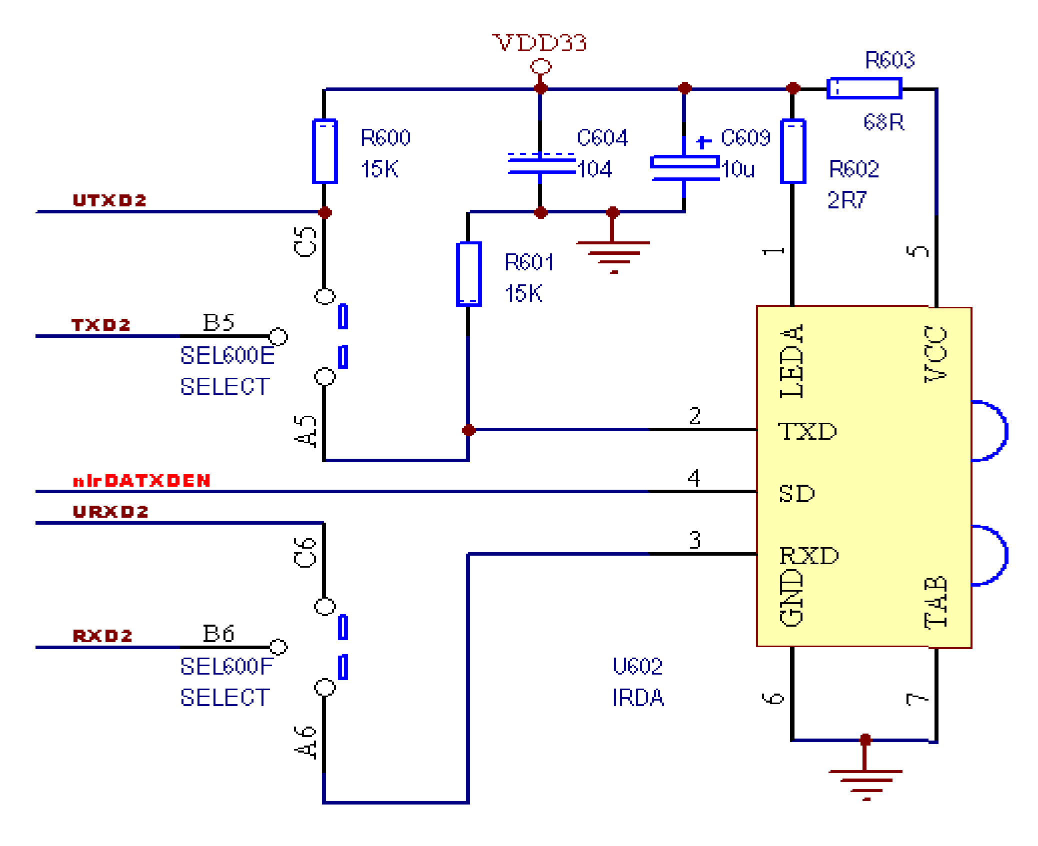


图 1-6

总线扩展口

为了方便用户可以再额外扩展设备，YFARM9-EDU-I 主板预留了总线扩展口 CON703。CON703 是 1 个 16bit 宽度的多功能扩展口，它不但引出了 16bit 资料线、位址线、DMA 控制线、中断、总线控制和复位信号。CON703 的引脚定义如下：

表 1-10

管脚	功能	管脚	功能	管脚	功能	管脚	功能
1	GND	12	DATA9	23	ADDR3	34	nWE
2	DATA0	13	DATA10	24	ADDR4	35	nOE
3	DATA1	14	DATA11	25	ADDR5	36	nXDREQ1
4	DATA2	15	DATA12	26	ADDR6	37	nXDACK1
5	DATA3	16	DATA13	27	ADDR7	38	nWAIT
6	DATA4	17	DATA14	28	GND	39	nGCS4
7	DATA5	18	DATA15	29	nRESET	40	IRQ_PCMIA
8	DATA6	19	GND	30	nGCS5	41	GND
9	DATA7	20	ADDR0	31	nWBE0	42	nIRQ_PCMIA
10	GND	21	ADDR1	32	nWBE1	43	VDD33
11	DATA8	22	ADDR2	33	GND	44	VDD33

USB 埠

S3C2410 内置 2 信道 USB HOST 端口，其中 1 个 HOST 端口可以配置为 USB Device（Slave）。下图是其中可配置 USB 端口部分的原理图。

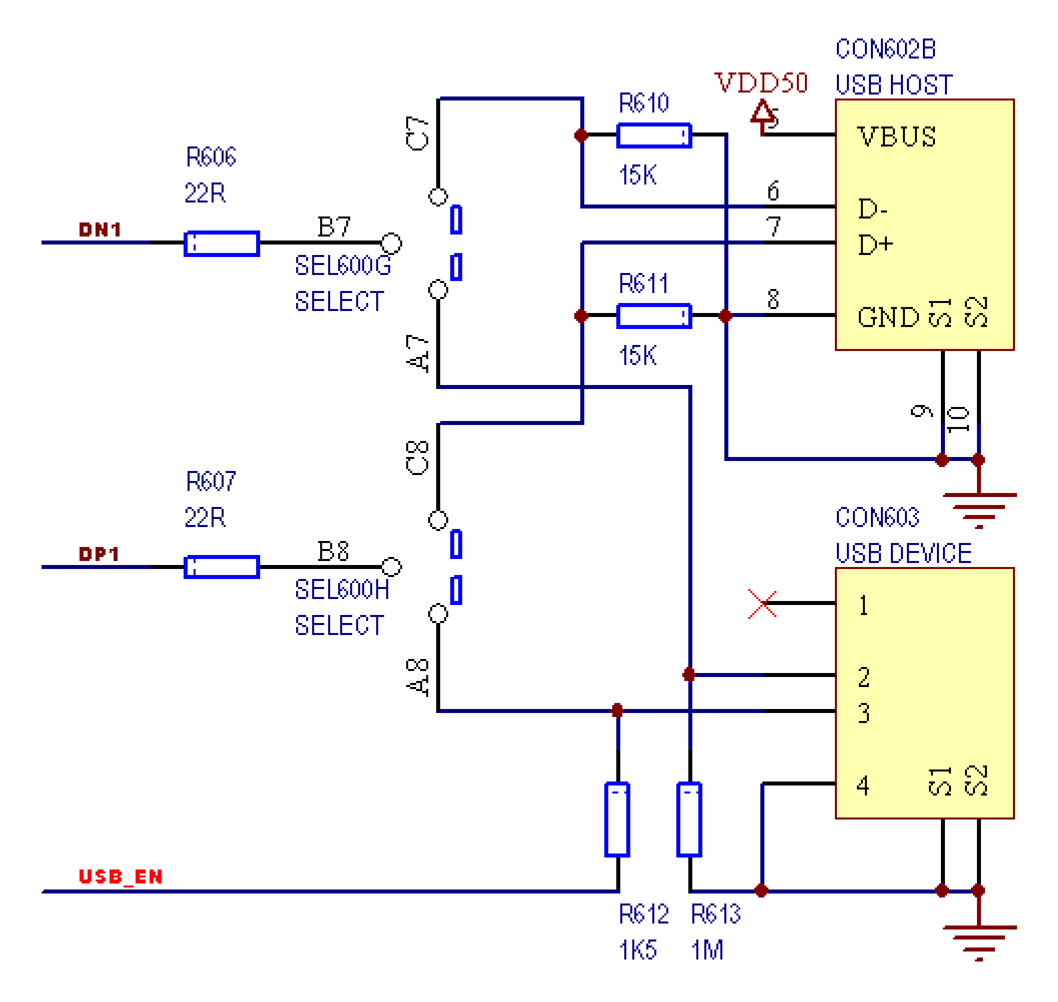


图 1-6

下表所示表明了如何将该 USB 端口配置为 HOST 或 DEVICE。

表 1-11

引脚功能	SEL600G		SEL600H		功能描述
	A-B	B-C	A-B	B-C	
USB1	C	0	C	0	USB DEVICE
	0	C	0	C	USB HOST

第二章 ARM 及开发工具简介

一、关于 ARM 处理器

ARM 公司自 1990 年正式成立以来，在 32 位 RISC (Reduced Instruction Set Computer CPU 开发领域不断取得突破，其结构已经从 V3 发展到 V6。由于 ARM 公司自成立以来，一直以 IP (Intelligence Property) 提供者的身份向各大半导体制造商出售知识产权，而自己从不介入芯片的生产销售，加上其设计的芯核具有功耗低、成本低等显著优点，因此获得众多的半导体厂家和整机厂商的大力支持，在 32 位嵌入式应用领域获得了巨大的成功，目前已经占有 75% 以上的 32 位嵌入式产品市场。在低功耗、低成本的嵌入式应用领域确立了市场领导地位。现在设计、生产 ARM 芯片的国际大公司已经超过 50 多家，国内中兴通讯和华为通讯等公司也已经购买 ARM 公司的芯核用于通讯专用芯片的设计。此外， ARM 芯片还获得了许多实时操作系统 (Real Time Operating System) 供货商的支持，比较知名的有： WINCE、Linux、pSOS、VxWorks 、Nucleus、EPOC、uCOS 等。

ARM 微处理器内核已经发展出了以下几个版本：

ARM7 系列：ARM7TDMI，ARM720T，ARM740T
ARM9 系列：ARM9TDMI，ARM920T，ARM926EJ
ARM10 系列：ARM1020E
StrongARM/Xscale 系列：SA1110/PXA255

常见的 ARM 处理器生产厂家：

Samsung（三星）
Intel（英特尔）
Amtel（爱特梅尔）
Cirrus Logic（凌云逻辑）
Sharp（夏普）
OKI（冲电子）
Motorola（摩托罗拉）

ARM 体系结构：

V4 结构：32 位寻址

T ：Thumb 状态：16 位元指令。
M ：长乘法支持（32*32=>64 或者 32*32+64=>64）。
D ：对调试的支持（Debug）
I ：嵌入的 ICE（In Circuit Emulation）

属于 V4 体系结构的处理器（核）有 ARM7， ARM7100（ ARM7 核的处理器），ARM7500 (ARM7 核的处理器)。

属于 V4T（支持 Thumb 指令）体系结构的处理器（核）有 ARM7TDMI，ARM7TDMI-S（ARM7TDMI 可综合版本），ARM710T（ARM7TDMI 核的处理器），ARM720T（ARM7TDMI 核的处理器），ARM740T（ARM7TDMI 核的处理器），ARM9TDMI，ARM910T（ARM9TDMI 核的处理器），ARM920T（ARM9TDMI 核的处理器），ARM940T（ARM9TDMI 核的处理器），StrongARM（Intel 公司的产品）。

V5 结构：提升了 ARM 和 Thumb 指令的交互工作能力。

E ：DSP 指令支持。
J ：Java 指令支持。

属于 V5T（支持 Thumb 指令）体系结构的处理器（核）有 ARM10TDMI，ARM1020T

（ARM10TDMI 核处理器）。

属于 V5TE（支持 Thumb ， DSP 指令）体系结构的处理器（核）有 ARM9E，ARM9E-S

（ARM9E 可综合版本），ARM946（ARM9E 核的处理器），ARM966（ARM9E 核的处理器），ARM10E，

ARM1020E（ARM10E 核处理器），ARM1022E（ARM10E 核的处理器）， Xscale（Intel 公司产

品）。

属于 V5TEJ（支持 Thumb，DSP 指令，Java 指令）体系结构的处理器（核）有 ARM9EJ，

ARM9EJ-S（ARM9EJ 可综合版本），ARM926EJ（ARM9EJ 核的处理器），ARM10EJ。

V6 结构：增加了媒体指令属于 V6 体系结构的处理器核有 ARM11。ARM 体系结构中有四种特殊指令集：Thumb 指令（T），DSP 指令（E），Java 指令（J），Media 指令，V6 体系结构包含全部四种特殊指令集。为满足向后兼容，ARMv6 也包括了 ARMv5 的内存管理和例外处理。这将使众多的第三方发展商能够利用现有的成果，支持软件和设计的复用种领域，比如嵌入控制、消费/教育类多媒体、DSP 和移动式应用等。

表 1-1 是各种 ARM 内核版本的性能对照

ARM CPU CORES	Cache Size (Inst/Data)	Tightly Coupled Memory	Memory Mgt	Bus Interface	Thumb	DSP	Jazelle
APPLICATION CORES							
ARM1020E	32k/32k	-	MMU	2x AHB	Yes	Yes	No
ARM1022E	16k/16k	-	MMU	2x AHB	Yes	Yes	No
ARM1026EJ-S	Variable	Yes	MMU or MPU	2x AHB	Yes	Yes	Yes
ARM1136J(F)-S	Variable	Yes	MMU	5x AHB	Yes	Yes	Yes
ARM1176JZ(F)-S	Variable	Yes	MMU + TrustZone	4x AXI	Yes	Yes	Yes
ARM720T	8k unified	-	MMU	AHB	Yes	No	No
ARM920T	16k/16k	-	MMU	ASB	Yes	No	No
ARM922T	8k/8k	-	MMU	ASB	Yes	No	No
ARM926EJ-S	Variable	Yes	MMU	2x AHB	Yes	Yes	Yes
MPCore Multiprocessor Core	Variable	-	MMU + cache coherency	1x or 2x AMBA AXI	Yes	Yes	Yes
EMBEDDED CORES							
ARM1026EJ-S	Variable	Yes	MMU or MPU	2x AHB	Yes	Yes	Yes
ARM1156T2(F)-S	Variable	Yes	MPU	4xAXI	Yes	Yes	No
ARM7EJ-S	-	-	-	Yes	Yes	Yes	Yes
ARM7TDMI	-	-	-	Yes**	Yes	No	No
ARM7TDMI-S	-	-	-	Yes	Yes	No	No
ARM946E-S	Variable	Yes	MPU	AHB	Yes	Yes	No
ARM966E-S	-	Yes	-	AHB	Yes	Yes	No
ARM968E-S		Yes	DMA	AHB-Lite	Yes	Yes	No
SECURE APPLICATIONS							
SecurCore SC100	-	-	MPU	-	Yes	No	No
SecurCore SC110	-	-	MPU	-	Yes	No	No
SecurCore SC200	-	-	MPU	-	Yes	Yes	Yes
SecurCore SC210	-	-	MPU	-	Yes	Yes	Yes

表 1-1

下图是 ARM 各种体系结构的特性对照表。

Architecture	Thumb®	DSP	Jazelle	Media	TrustZone	Thumb-2
v4T	✔					
v5TE	✔	✔				
v5TEJ	✔	✔	✔			
v6	✔	✔	✔	✔		
v6Z	✔	✔	✔	✔	✔	
v6T2	✔	✔	✔	✔		✔

表 1-2

二、关于 ARM 的操作系统

采用ARM内核的处理器由于性能强大，因此能支持绝大多数的嵌入式操作系统：

uCOS

Nucleus

VxWorks

SuperTask

Linux

WinCE.net

ECOS等等

尤其是Microsoft已经宣布，今后的WinCE.net操作系统的PPC（掌上计算机）版本只支持基于ARM的处理器，这更说明了ARM处理器在操作系统方面的优势。

三、关于 ARM 开发工具

ARM开发过程中所使用的开发工具主要分3类。分别是集成开发环境（IDE）、在线仿真器（ICE）和实验平台（DVK/EVK）

目前业界主流的集成开发环境有：

ARM公司 ： Arm Developer Suite1.2(简称ADS1.2)

IAR ： Embedded Workbench

GreenHills ： Multi 2000

免费的IDE ： GCC

但是由于ARM公司的ADS1.2版本由于编译效率高，使用方便，而成为广大ARM开发工程师的首选。

主流的基于JTAG的仿真器有：

ARM ： Multi-ICE/Multi-Trace

Abatron AG ： BDI1000/BDI2000

EPI ： MAJIC/JEENI

Lauterbach ： Trace32-ICD

同样 ARM 公司的 Multi-ICE 系列仿真器由于支持的 CPU 内核种类最为全面，并且性价适中而得到广泛的使用。

第三章 LCD 控制实验

一、超薄平面显示器时代来临

电视机所采用的 CRT(阴极射线管)有着体积大、重量重、尺寸受限等缺点。随着电子科技的发展，对移动显示的要求越来越多，CRT 的先天限制，让其小型化、行动化的理想受到阻碍。这使得开发新一代的显示器技术变得更有其必要！

新一代的显示器讲求几个重点：平面直角，画面显示不变形、轻薄短小耗能少，携带方便且同时要与现有的影像信号技术兼容。目前谈论到超薄型显示器技术，最普及当是 TFT LCD 的应用了，举凡数字相机、笔记型计算机、PDA 等，需要显示复杂信息的电子产品通通少不了它。TFT LCD 技术又包含了，低温多硅晶TFT LCD、反射式TFT LCD 等，多项不同的显示技术，下面我们就要来一探 LCD 的历史与原理。

二、液晶的发明与发现

液晶的诞生来自于一项非常特殊物质的发现，早在 1850 年 Virchow, Mettenheimer 和 Valentin 这三个人就发现 nerve fibre 的粹取物中含有这种不寻常的东西。到了 1877 年德国物理学家 Otto Lehmann 运用偏极化的显微镜首次观测到了液晶化的现象，但他对此一现象的成因并不了解。直到公元1888年，奥地利的植物学家 Friedrich Reinitzer（1857-1927）发现了螺旋性甲苯酸盐的化合物（cholesteryl benzoate），确认了这种化合物在加热时具有两个不同温度的熔点，在这两个不同的温度点中，其状态介于一般液态与固态物质之间，类似胶状，但在某一温度范围内其又具有液体和结晶双方性质，由于其特殊的状态。Reinitzer 后来走访 Lehmann 深入探讨这种物质的表现，其后两人便命名这种物质为「Liquid Crystal」，就是液态结晶物质的意思。Reinitzer 和 Lehmann 这两人被誉为液晶之父。

同 CRT 阴极射线管一样，液晶虽早在1888年就被发现（实际上，但是实际应用在生活周遭时，已是80年后的事了。因为液晶在两次大战中对军事用途的帮助不大，以致于 其发展落后 CRT 甚多。比较重要的是 1922 年 Oseen 和 Zöcher 这两位科学家为液晶确立状态变化之方程式。一直到了 1968年美国RCA公司工程师们利用液晶分子受到电压的影响而改变其分子的排列状态，并且可以让入射光线产生偏转的现象之原理，制造了世界第一台使用液晶显示的屏幕。由此开始，加上了1970年代日本 SONY 与 Sharp 两家公司对液晶显示技术全面开发与应用，让液晶显示器成功的融入现代的电子产品之中。

描述液晶的物理性质，必须先了解一般固态晶体具有方向性，而液态晶体这种特殊物质，不但具有一般固体晶体的方向性外，同时又具有液体的流动性。改变固态晶体方向必须旋转整个晶体，改变液态晶体就不用那么麻烦，它的方向性可经由电场或磁场来控制。

改变液晶的方向视液晶的成分而有所不同，有的液晶和电场平行时位能较低，所以当外加电场时会朝着电场方向转动，相对的，也有液晶是对应电场垂直时位能较低。由于液晶对于外加力量（电场或磁场敏感），从而呈现了方向性的效果，也导致了当光线入射液晶中时，必然会按照液晶分子的排列方式行进，产生了自然的偏转现像（见图3-1提供）。

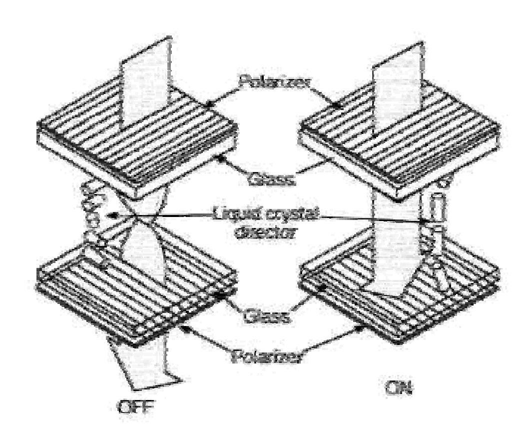
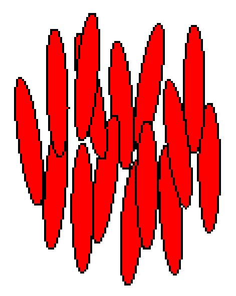


图3-1

部分液晶分子的电子结构中，有着很强的电子共轭运动能力，所以当液晶分子受到外加电场的作用，便很容易的被极化产生感应偶极性（induced dipolar），这也是液晶分子之间互相作用力量的来源。而一般电子产品中所用的液晶显示器，就是是利用液晶的光电效应，藉由外部的电压控制，再透过液晶分子的折射特性，以及对光线的旋转能力来获得亮暗情况，进而达到显像的目的。



电源关闭时，液晶具有偏光效果
可将入射光线转弯，穿过极栅，呈现亮色



电源开启时液晶不具有偏光的功能
因此光线不能通过极栅呈现暗色

三、液晶显示器的种类

利用液晶制成的显示器称为液晶显示器，英文称 LCD（Liquid Crystal Display）。其种类可分为依驱动方式之静态驱动（Static）、单纯矩阵驱动（Simple Matrix）以及主动矩阵驱动（Active Matrix）三种。而其中，单纯矩阵型又是俗称的被动式（Passive），可分为扭转向列型（Twisted Nematic，简称 TN）和超扭转式向列型（Super Twisted Nematic，简称STN）两种；而主动矩阵型则以薄膜式晶体管型（Thin Film　Transistor；TFT）为目前主流。

TN型

TN型液晶显示技术可说是液晶显示器中最基本的，其它种类的液晶显示器也可说是以TN型为蓝本加以改良。同样的，它的运作原理也较其它技术来的简单。TN 的构造包括了垂直方向与水平方向的偏光板（Polarizer），其上具有细纹沟槽，中间夹杂液晶材料以及导电的玻璃基板（Glass）。

STN/DSTN

STN型的显示原理也类似，不同的是TN型的液晶分子是将入射光旋转90度，而STN则可将入射光旋转180~270度。 单纯的 TN 显示器本身只有明暗两种显示（或黑白），无法产生色彩的变化。TN LCD 采用的是“直接驱动”无法显示较多的像素，且画面的对比小，反应速度慢，视角更仅在+30度以下(即观赏角度约60度)，显示质量也较差；故TN型LCD主要用途在于简单的数字元与文字的显示，如：电子表及电子计算器等。 STN的出现改善了

视角狭小的缺点并提高对比率，STN以“多任务驱动”增加扫瞄线数提高画素显示，品质较TN来得高。再搭配彩色滤光片的使用，将单色显示矩阵的任一像素（pixel）分成三个子像素（sub-pixel），分别透过彩色滤光片显示红、绿、蓝三原色，再经由三原色比例之调和，可以显示出逼近全彩模式的色彩。由于 STN 显示的画面色彩对比度仍只达30：1(对比愈小，画面愈不清楚)；反应速度为150ms（毫秒），作为一般操作显示接口尚可，但若要播放电影速度仍然不够。

由于 STN 仍有不少缺点，后续的 DSTN则通过双扫描方式来显示，由于DSTN采用双扫描技术，因此显示效果相对STN来说，有大幅度提高。DSTN 反应速度可达到 100ms，但因它们都为“被动式驱动”，在电场反复改变电压的过程中，每一像素的恢复过程都较慢，在屏幕画面快速变化时，例如：显示网球比赛的转播，就会产生所谓的“拖尾”现象。特别是当网球选手击球的那一瞬间，你就可以看到拖屏幕上出现“球迹尾”现象。不过，DSTN价格便宜、功耗能低，一些 PDA 等，仍使用 DSTN 作为显示装置。

TFT

TN与STN型液晶显示器都是使用场电压驱动方式，如果显示尺寸加大，中心部位对电极变化的反应时间就会拉长，显示器的速度就跟不上。 为了改善这个的问题，主动式矩阵（active-matrix ）驱动被提出，主动式 TFT型的液晶显示器的结构较为复杂包括了：背光管、导光板、偏光板、滤光板、玻璃基板、配向膜、液晶材料和薄模式晶体管等等（如图3-2）。在TFT型液晶显器中，导电玻璃上画上网状的细小线路，电极则由是薄膜式晶体管所排列而成的矩阵开关，在每个线路相交的地方配有控制闸，各显示点控制闸配合驱动讯号作动。电极上之晶体管矩阵依显示讯号开启或关闭液晶分子的电压，使液晶分子轴转向而成“亮”或“暗”的对比，避免了显示器对电场效应的依靠，转以晶体管开启和关闭的速率作为决定步骤。 也因此，TFT-LCD 的显示质量较 TN/STN佳，画面显示对比可达150：1以上，反应速度逼近 30ms 甚至更快。同时又可以全彩甚至真彩效果显示，产品适用于PDA、笔记型计算机、液晶显示器、汽车导航系统、数字相机及液晶投影机。

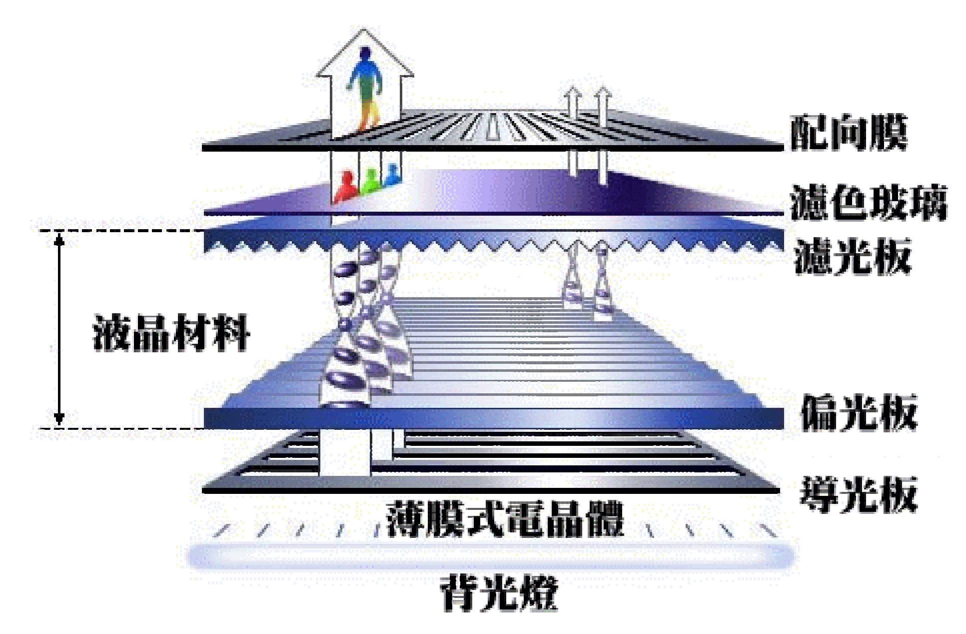


图3-2

下面的表格对TN、STN、TFT的各自特性做了对比

TN、STN 及 TFT 型液晶显示器之比较表			
类别	**TN**	**STN**	**TFT**
原理	液晶分子，扭转 90 度	扭转 180~270 度	液晶分子，扭转 90 度

特性	黑白、单色 低对比（20：1）	黑白、彩色（26 万色） 低对比，较 TN 佳 （40：1）	彩色（1667 万色） 高对比，较 STN 佳（300：1）
全色彩化	否	否	可媲美 CRT 之全彩色
动画显示	否	否	可媲美 CRT
视角	30 度以下	40 度以下	80 度以下
面板尺寸	1~3 寸	1~12 寸	6~17 寸以上
应用范围	电子表、计算器	电子字典、行动电话	彩色笔记本计算机、投影机、超薄平面彩色电视

四、液晶显示器的发展与未来

TFT LCD 之所以成功，在于其每个像素后面都配置一个晶体管开关作为控制整合之用，以致于整个 TFT LCD 看起来就类似一个大型整合电路。由于 TFT LCD 必须将画素作得非常小，让人眼只能看到画面，分辨不出画素，所以 TFT LCD 的生产工艺就相当精密。过去，因为技术尚未成熟，在一大片的 TFT LCD 当中难免有些节点，无法连接或连接错误，导致无法显示正确画素，这些统称“坏点”，包含常见的“红、蓝、绿点”无法自行控制、“黑、白点”无法使用等。目前高精密的技术已经足以克服 TFT LCD 在生产过程中产生“坏点”的机率，部分“坏点”也可通过“暗点化”（人类的眼睛对于暗画素不敏感）将其消隐。

由于 TFT-LCD 成功的解决 CRT 的缺点，连带的使其应用范围加广范！同时，也发生了一些意想不到的问题，例如：在阳光下 TFT LCD 显示不佳，需要倚靠遮光罩或透光式设计减少反光的发生，才能将其看得清楚。另外，也有利用特殊镀膜技术，减少背景光泄漏、增加屏幕黑度、提高对比度的作用，并可以同时减小在日常明亮工作环境下的眩光现象。

五、S3C2410 内置 LCD 控制器详解

一块LCD屏显示图像，不但需要LCD驱动器，还需要有相应的LCD控制器。通常LCD驱动器会以COF/COG的形式与LCD 玻璃基板制做在一起，而LCD控制器则有外部电路来实现。而S3C2410内部已经集成了LCD控制器，因此可以很方便地去控制各种类型的LCD屏，例如：STN和TFT屏。由于TFT屏将是今后应用的主流，因此接下来，重点围绕TFT屏的控制来进行。

S3C2410 LCD控制器的特性：

STN屏

－支持3种扫描方式：4bit单扫、4位双扫和8位单扫

－支持单色、4级灰度和16级灰度屏

－支持256色和4096色彩色STN屏（CSTN）

－支持分辩率为640*480、320*240、160*160以及其它规格的多种LCD

TFT屏

－支持单色、4级灰度、256色的调色板显示模式

－支持64K和16M色非调色板显示模式

－支持分辩率为640*480，320*240及其它多种规格的LCD

对于控制TFT屏来说，除了要给它送视频资料（VD[23:0]）以外，还有以下一些信号是必不可少的，分别是：

VSYNC（VFRAME） ：帧同步信号

HSYNC（VLINE） ：行同步信号

VCLK ：像数时钟信号

VDEN（VM） ：数据有效标志信号

图3-3是S3C2410内部的LCD控制器的逻辑示意图：

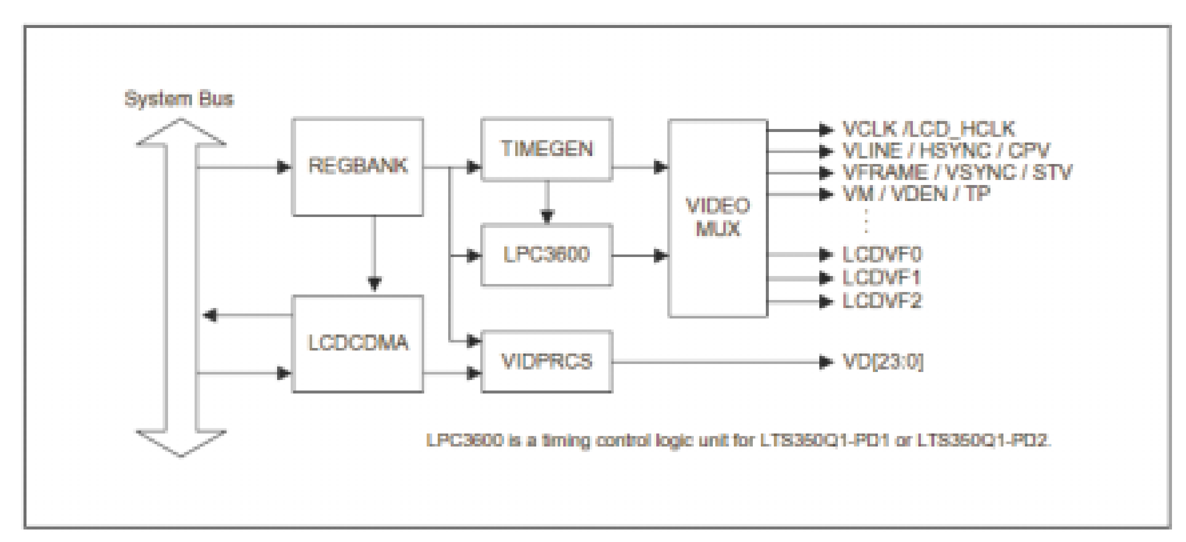


图3-3

REGBANK 是LCD控制器的寄存器组，用来对LCD控制器的各项参数进行设置。而LCDCDMA 则是LCD控制器专用的DMA信道，负责将视频资料从系统总线（System Bus）上取来，通过 VIDPRCS 从VD[23:0]发送给LCD屏。同时 TIMEGEN 和 **LPC3600** 负责产生 LCD屏所需要的控制时序，例如VSYNC、HSYNC、VCLK、VDEN，然后从 VIDEO MUX 送给LCD屏。

TFT屏时序分析

图3-4是TFT屏的典型时序。其中VSYNC是帧同步信号，VSYNC每发出1个脉冲，都意味着新的1屏视频资料开始发送。而HSYNC为行同步信号，每个HSYNC脉冲都表明新的1行视频资料开始发送。而VDEN则用来标明视频资料的有效，VCLK是用来锁存视频资料的像数时钟。

并且在帧同步以及行同步的头尾都必须留有回扫时间，例如对于VSYNC来说前回扫时间就是（VSPW+1）＋（VBPD+1），后回扫时间就是（VFPD+1）；HSYNC亦类同。这样的时序要求是当初CRT显示器由于电子枪偏转需要时间，但后来成了实际上的工业标准，乃至于后来出现的TFT屏为了在时序上于CRT兼容，也采用了这样的控制时序。

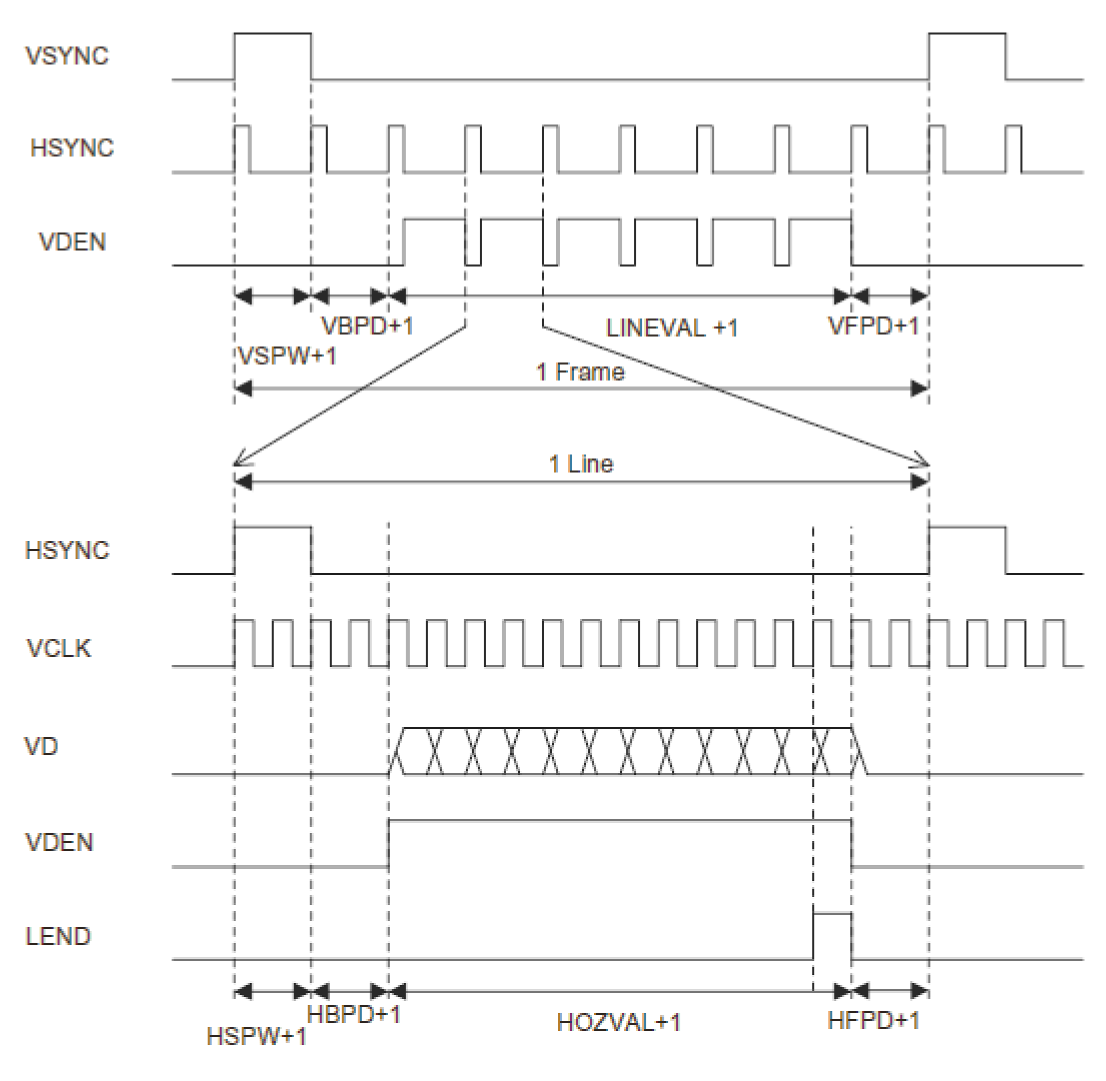

VSYNC
HSYNC
VDEN
VBPD+1
LINEVAL +1
VFPD+1
VSPW+1
1 Frame
1 Line
HSYNC
VCLK
VD
VDEN
LEND
HBPD+1
HOZVAL+1
HFPD+1
HSPW+1

图3-4

YFARM9-EDU-1采用的是Samsung公司的1款3.5寸TFT真彩LCD屏，分辨率为240*320，下图为该屏的时序要求。

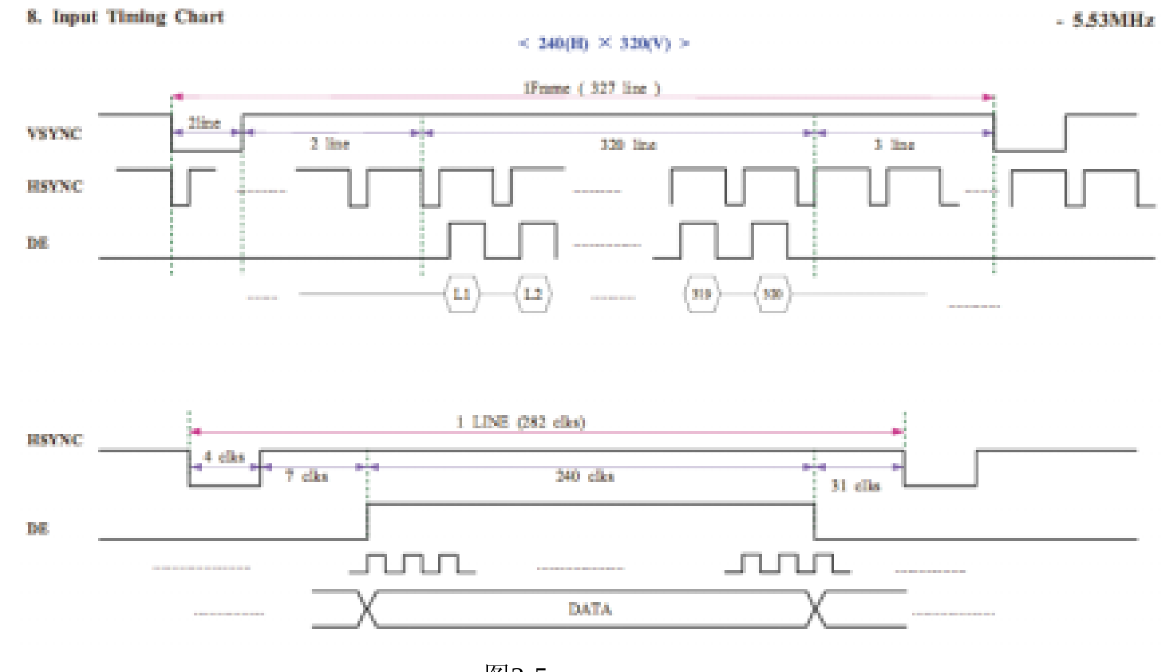


图3-5

通过对比图3-4和图3-5，我们不难看出：

VSPW+1=2 -> VSPW=1

VBPD+1=2 -> VBPD=1

LINVAL+1=320-> LINVAL=319

VFPD+1=3 -> VFPD=2

HSPW+1=4 -> HSPW=3

HBPD+1=7 -> HBPW=6

HOZVAL+1=240-> HOZVAL=239

HFPD+1=31 -> HFPD=30

以上各参数，除了LINVAL和HOZVAL直接和屏的分辨率有关，其它的参数在实际操作过程中应以上面的为参考，不应偏差太多。

LCD控制器主要寄存器功能详解

(1)LCDCON1

LCDCON1	Bit	Description	Initial State
LINECNT (read only)	[27:18]	Provide the status of the line counter. Down count from LINEVAL to 0	0000000000
CLKVAL	[17:8]	Determine the rates of VCLK and CLKVAL[9:0]. **STN**: VCLK = HCLK / (CLKVAL × 2) (CLKVAL ≥ 2) **TFT**: VCLK = HCLK / [(CLKVAL+1) × 2] (CLKVAL ≥ 0)	0000000000
MMODE	[7]	Determine the toggle rate of the VM. 0 = Each Frame, 1 = The rate defined by the MVAL	0
PNRMODE	[6:5]	Select the display mode. 00 = 4-bit dual scan display mode (STN) 01 = 4-bit single scan display mode (STN) 10 = 8-bit single scan display mode (STN) 11 = TFT LCD panel	00
BPPMODE	[4:1]	Select the BPP (Bits Per Pixel) mode. 0000 = 1 bpp for STN, Monochrome mode 0001 = 2 bpp for STN, 4-level gray mode 0010 = 4 bpp for STN, 16-level gray mode 0011 = 8 bpp for STN, color mode 0100 = 12 bpp for STN, color mode 1000 = 1 bpp for TFT 1001 = 2 bpp for TFT 1010 = 4 bpp for TFT 1011 = 8 bpp for TFT 1100 = 16 bpp for TFT 1101 = 24 bpp for TFT	0000
ENVID	[0]	LCD video output and the logic enable/disable. 0 = Disable the video output and the LCD control signal. 1 = Enable the video output and the LCD control signal.	0

LINECNT ：当前行扫描计数器值，标明当前扫描到了多少行

CLKVAL：决定VCLK的分频比。LCD控制器输出的VCLK是直接由系统总线（AHB）的工作频率HCLK直接分频得到的。做为240*320的TFT屏，应保证得出的VCLK在5~10MHz之间

MMODE：VM信号的触发模式（仅对STN屏有效，对TFT屏无意义）

PNRMODE：选择当前的显示模式，对于TFT屏而言，应选择[11]，即TFT LCD panel

BPPMODE：选择色彩模式，对于真彩显示而言，选择16bpp（64K色）即可满足要求

ENVID：使能LCD信号输出

LCDCON2	Bit	Description	Initial State
VBPD	[31:24]	**TFT**: Vertical back porch is the number of inactive lines at the start of a frame, after vertical synchronization period. **STN**: These bits should be set to zero on STN LCD.	0x00
LINEVAL	[23:14]	**TFT/STN**: These bits determine the vertical size of LCD panel.	0000000000
VFPD	[13:6]	**TFT**: Vertical front porch is the number of inactive lines at the end of a frame, before vertical synchronization period. **STN**: These bits should be set to zero on STN LCD.	00000000
VSPW	[5:0]	**TFT**: Vertical sync pulse width determines the VSYNC pulse's high level width by counting the number of inactive lines. **STN**: These bits should be set to zero on STN LCD.	000000

VBPD，LINEVAL，VFPD，VSPW 的各项含义已经在前面的时序图中得到体现，这里不再赘述。

LCDCON3	Bit	Description	Initial state
HBPD (TFT)	[25:19]	**TFT**: Horizontal back porch is the number of VCLK periods between the falling edge of HSYNC and the start of active data.	0000000
WDLY (STN)		**STN**: WDLY[1:0] bits determine the delay between VLINE and VCLK by counting the number of the HCLK. WDLY[7:2] are reserved. 00 = 16 HCLK, 01 = 32 HCLK, 10 = 48 HCLK, 11 = 64 HCLK	
HOZVAL	[18:8]	**TFT/STN**: These bits determine the horizontal size of LCD panel. HOZVAL has to be determined to meet the condition that total bytes of 1 line are 4n bytes. If the x size of LCD is 120 dot in mono mode, x=120 cannot be supported because 1 line consists of 15 bytes. Instead, x=128 in mono mode can be supported because 1 line is composed of 16 bytes (2n). LCD panel driver will discard the additional 8 dot.	00000000000
HFPD (TFT)	[7:0]	**TFT**: Horizontal front porch is the number of VCLK periods between the end of active data and the rising edge of HSYNC.	0X00
LINEBLANK (STN)		**STN**: These bits indicate the blank time in one horizontal line duration time. These bits adjust the rate of the VLINE finely. The unit of LINEBLANK is HCLK X 8. Ex) If the value of LINEBLANK is 10, the blank time is inserted to VCLK during 80 HCLK.	

HBPD ，HOZVAL ，HFPD 的各项含义已经在前面的时序图中得到体现，这里不再赘述。

LCDCON4	Bit	Description	Initial state
MVAL	[15:8]	**STN**: These bit define the rate at which the VM signal will toggle if the MMODE bit is set to logic '1'.	0X00
HSPW(TFT)	[7:0]	**TFT**: Horizontal sync pulse width determines the HSYNC pulse's high level width by counting the number of the VCLK.	0X00
WLH(STN)		**STN**: WLH[1:0] bits determine the VLINE pulse's high level width by counting the number of the HCLK. WLH[7:2] are reserved. 00 = 16 HCLK, 01 = 32 HCLK, 10 = 48 HCLK, 11 = 64 HCLK	

HSPW 的含义已经在前面的时序图中得到体现，这里不再赘述。

MVAL 只对 STN屏有效，对TFT屏无意义。

LCDCON5	Bit	Description	Initial state
Reserved	[31:17]	This bit is reserved and the value should be '0'.	0
VSTATUS	[16:15]	TFT: Vertical Status (read only). 00 = VSYNC 01 = BACK Porch 10 = ACTIVE 11 = FRONT Porch	00
HSTATUS	[14:13]	TFT: Horizontal Status (read only). 00 = HSYNC 01 = BACK Porch 10 = ACTIVE 11 = FRONT Porch	00
BPP24BL	[12]	TFT: This bit determines the order of 24 bpp video memory. 0 = LSB valid 1 = MSB Valid	0
FRM565	[11]	TFT: This bit selects the format of 16 bpp output video data. 0 = 5:5:5:1 Format 1 = 5:6:5 Format	0
INVVCLK	[10]	STN/TFT: This bit controls the polarity of the VCLK active edge. 0 = The video data is fetched at VCLK falling edge 1 = The video data is fetched at VCLK rising edge	0
INVVLINE	[9]	STN/TFT: This bit indicates the VLINE/HSYNC pulse polarity. 0 = Normal 1 = Inverted	0
INVVFRAME	[8]	STN/TFT: This bit indicates the VFRAME/VSYNC pulse polarity. 0 = Normal 1 = Inverted	0
INVVD	[7]	STN/TFT: This bit indicates the VD (video data) pulse polarity. 0 = Normal 1 = VD is inverted.	0

VSTATUS：当前VSYNC信号扫描状态，指明当前VSYNC同步信号处于何种扫描阶段
HSTATUS：当前HSYNC信号扫描状态，指明当前HSYNC同步信号处于何种扫描阶段
BPP24BL：设定24bpp显示模式时，视频资料在显示缓冲区中的排列顺序（即低位元有效还是高位有效）。对于16bpp的64K色显示模式，该设置位元无意义。
FRM565：对于16bpp显示模式，有2中形式，一种是RGB＝5:5:5:1，另一种是5:6:5。后一种模式最为常用，它的含义是表示64K种色彩的16bit RGB资料中，红色（R）占了5bit，绿色（G）占了6bit，兰色（B）占了5bit
INVVCLK，INVLINE，INVFRAME，INVVD：通过前面的时序图，我们知道，CPU的LCD控制器输出的时序默认是正脉冲，而LCD需要VSYNC（VFRAME）、VLINE（HSYNC）均为负脉冲，因此INVLINE和INVFRAME必须设为“1 ”，即选择反相输出。

LCDCON5	Bit	Description	Initial state
INVVDEN	[6]	**TFT**: This bit indicates the VDEN signal polarity. 0 = Normal 1 = Inverted	0
INVPWREN	[5]	**STN/TFT**: This bit indicates the PWREN signal polarity. 0 = Normal 1 = Inverted	0
INVLEND	[4]	**TFT**: This bit indicates the LEND signal polarity. 0 = Normal 1 = Inverted	0
PWREN	[3]	**STN/TFT**: LCD_PWREN output signal enable/disable. 0 = Disable PWREN signal 1 = Enable PWREN signal	0
ENLEND	[2]	**TFT**: LEND output signal enable/disable. 0 = Disable LEND signal 1 = Enable LEND signal	0
BSWP	[1]	**STN/TFT**: Byte swap control bit. 0 = Swap Disable 1 = Swap Enable	0
HWSWP	[0]	**STN/TFT**: Half-Word swap control bit. 0 = Swap Disable 1 = Swap Enable	0

INVVDEN， INVPWREN， INVLEND 的功能同前面的类似。
PWREN 为LCD电源使能控制。在CPU LCD控制器的输出信号中，有一个电源使能管脚LCD_PWREN，用来做为LCD屏电源的开关信号。
ENLEND 对普通的TFT屏无效，可以不考虑。
BSWP 和 HWSWP 为字节（Byte）或半字（Half-Word）交换使能。由于不同的 GUI 对 FrameBuffer（显示缓冲区）的管理不同，必要时需要通过调整 BSWP 和 HWSWP 来适应GUI。

第四章　LED 及键盘驱动实验

一、LED 的结构及发光原理

50年前人们已经了解半导体材料可产生光线的基本知识，第一个商用发光二极管产生于1960年。LED是英文light emitting diode（发光二极管）的缩写，它的基本结构是一块电致发光的半导体材料，置于一个有引线的架子上，然后四周用环氧树脂密封，起到保护内部芯线的作用，所以LED的抗震性能好。

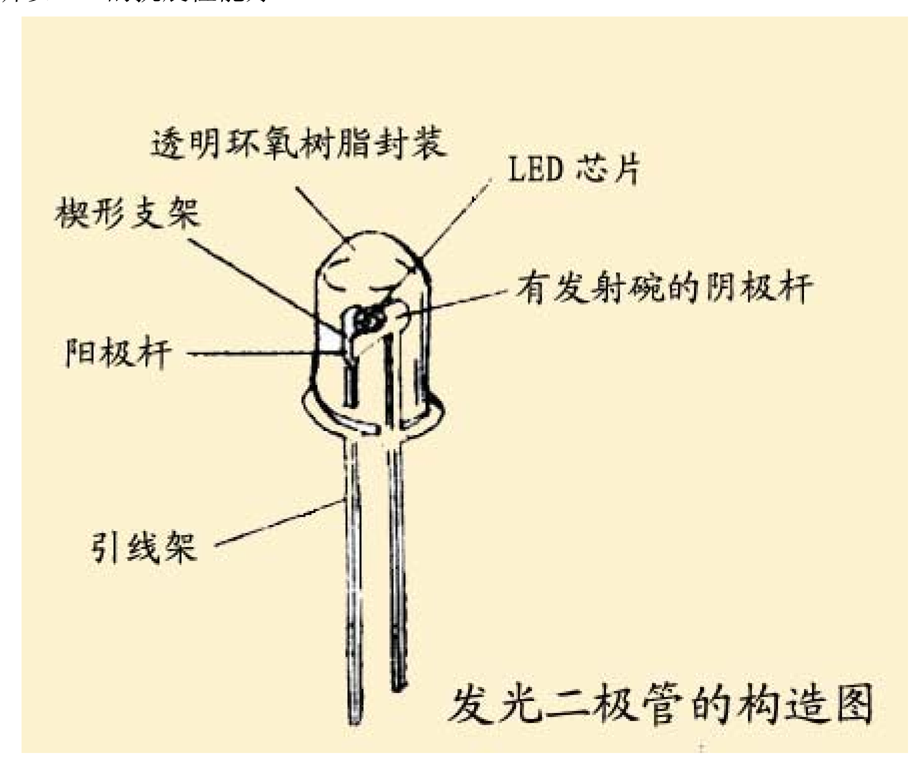


图4-1

LED结构图如图4-1所示。发光二极管的核心部分是由p型半导体和n型半导体组成的芯片，在p型半导体和n型半导体之间有一个过渡层，称为p-n结。在某些半导体材料的PN结中，注入的少数载流子与多数载流子复合时会把多余的能量以光的形式释放出来，从而把电能直接转换为光能。PN结加反向电压，少数载流子难以注入，故不发光。这种利用注入式电致发光原理制作的二极管叫发光二极管，通称LED。 当它处于正向工作状态时（即两端加上正向电压），电流从LED阳极流向阴极时，半导体晶体就发出从紫外到红外不同颜色的光线，光的强弱与电流有关。

二、LED 光源的特点

1. 电压：LED使用低压电源，根据产品不同而异，所以它是一个比使用高压电源更安全的电源，特别适用于公共场所。
2. 效能：消耗能量较同光效的白炽灯减少80%
3. 适用性：很小，每个单元LED小片是3-5mm的正方形，所以可以制备成各种形状的器件，并且适合于易变的环境
4. 稳定性：10万小时，光衰为初始的50%
5. 响应时间：其白炽灯的响应时间为毫秒级，LED灯的响应时间为纳秒级
6. 对环境污染：无有害金属汞
7. 颜色：改变电流可以变色，发光二极管方便地通过化学修饰方法，调整材料的能带结构

和带隙，实现红黄绿兰橙多色发光。如小电流时为红色的LED，随着电流的增加，可以依次变为橙色，黄色，最后为绿色

8. 价格：LED的价格比较昂贵，较之于白炽灯，几只LED的价格就可以与一只白炽灯的价格相当，而通常每组信号灯需由上300～500只二极管构成。

三、单色光 LED 的种类及其发展历史

最早应用半导体P-N结发光原理制成的LED光源问世于20世纪60年代初。当时所用的材料是GaAsP，发红光（λ_p=650nm)，在驱动电流为20毫安培时，光通量只有千分之几个流明，相应的发光效率约0.1流明/瓦。

70年代中期，引入元素In和N，使LED产生绿光（λ_p=555nm)，黄光（λ_p=590nm）和橙光（λ_p=610nm)，光效也提高到1流明/瓦。

到了80年代初，出现了GaAlAs的LED光源，使得红色LED的光效达到10流明/瓦。

90年代初，发红光、黄光的GaAlInP和发绿、蓝光的GaInN两种新材料的开发成功，使LED的光效得到大幅度的提高。在2000年，前者做成的LED在红、橙区（λ_p=615nm）的光效达到100流明/瓦，而后者制成的LED在绿色区域（λ_p=530nm)的光效可以达到50流明/瓦。

四、单色光 LED 的应用

最初LED用作仪器仪表的指示光源，后来各种光色的LED在交通信号灯和大面积显示屏中得到了广泛应用，产生了很好的经济效益和社会效益。以12英寸的红色交通信号灯为例，在美国本来是采用长寿命，低光效的140瓦白炽灯作为光源，它产生2000流明的白光。经红色滤光片后，光损失90%，只剩下200流明的红光。而在新设计的灯中，Lumileds公司采用了18个红色LED光源，包括电路损失在内，共耗电14瓦，即可产生同样的光效。

汽车信号灯也是LED光源应用的重要领域。1987年，我国开始在汽车上安装高位刹车灯，由于LED响应速度快（纳秒级），可以及早让尾随车辆的司机知道行驶状况，减少汽车追尾事故的发生。

另外，LED灯在室外红、绿、蓝全彩显示屏，匙扣式微型电筒、仪器仪表等领域都得到了应用。

五、LED 的驱动

由于单只LED管的工作电压低（大约在1.5~2V)，个别需达到4V，同时工作电流仅为1~5mA，因此可以用CPU的通用输入输出管脚（GPIO）就可直接控制。

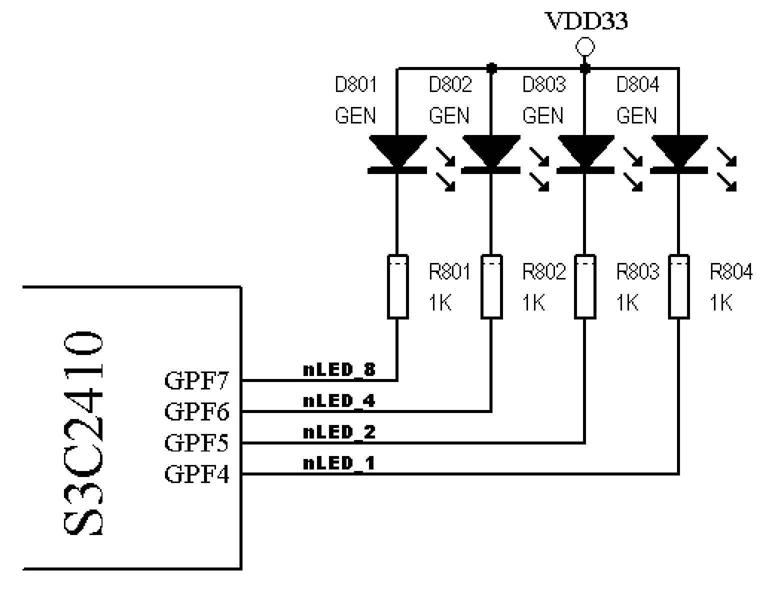


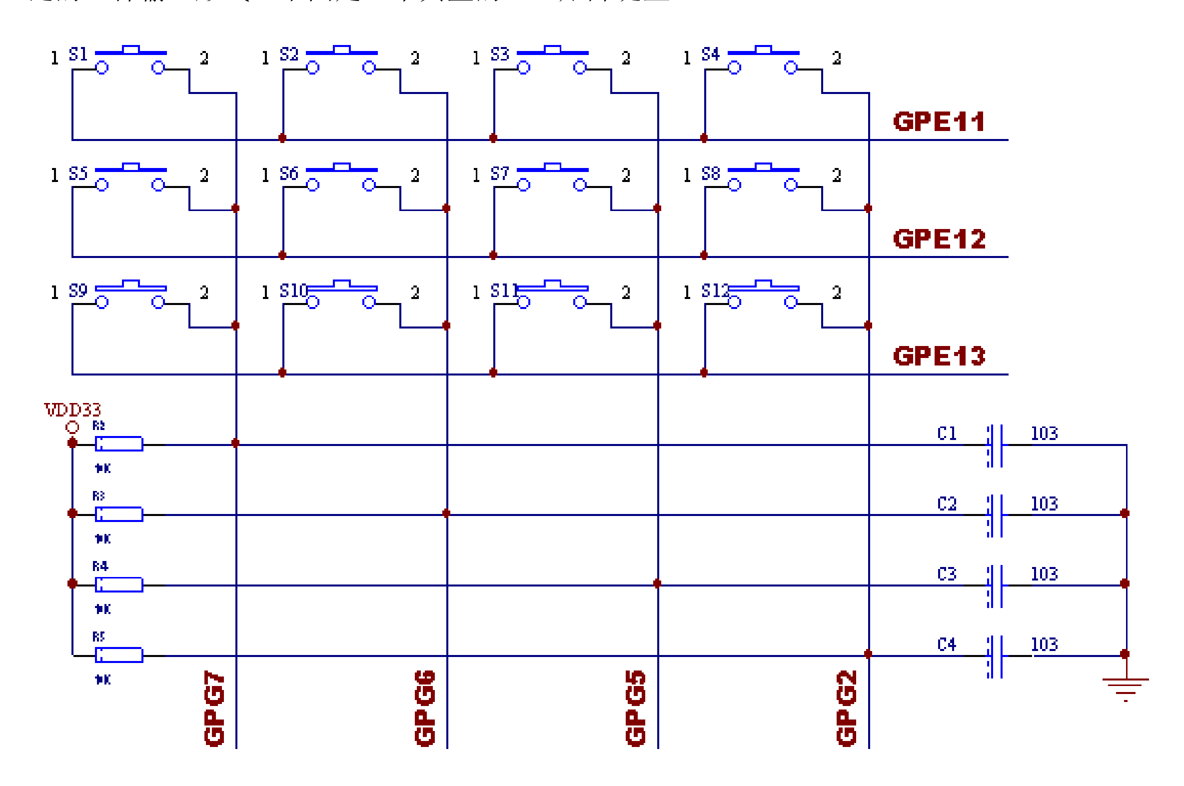
图4-2

图4-2是直接由CPU的GPIO直接驱动（控制LED）的原理图。以D804为例，它的阳极接在3.3V电源上，阴极通过1只串联的1K欧姆电阻接在CPU的GPF4 IO管脚上。假设D804的正向导通压降为1.5V，当GPF4输出低电平时（一般为0.2V左右），D804（绿色LED）中将会有（3.3V-0.2V-1.5V）/1K欧姆＝1.6mA的电流流过，并导致D804发光。反之GPF4输出高电平时，D804中几乎没有电流流过，因此将不会发光。

所以要控制LED的发光与否，关键在于我们如何去控制CPU的GPF埠的输出状态。

六、矩阵键盘的扫描

在一个完整的人机操作接口中，光有指示功能的LED，往往是不够的，还需要有输入设备。矩阵键盘由于电路简单，操作方便，同时占用的系统资源也很少，所以成为使用非常广泛的一种输入形式。下图是一个典型的3×4矩阵键盘：



图中 GPE 口做为扫描输出， GPE11、GPE12、GPE13依次轮流输出低电平，而 GPG 口则做为扫描输入口（以的外部中断，或查询输入口状态的形式）。若有键按下，即可知该键所在行、列，得出该键的键值。

注：图中电阻为外部上拉电阻，选值 10K，为的是有足够小的上来电阻，尽可能减少信号阻抗，防止干扰。图中的电容是为了滤波，滤除按键信号中的毛刺（这与按键抖动不同）。

七、S3C2410 内部 GPIO 概述

S3C2410共有117只多功能输入/输出管脚，它们分别是：

－GPA口：23只输出口

－GPB口：11只输入/输出口

－GPC口：16只输入/输出口

－GPD口：16只输入/输出口

－GPE口：16只输入/输出口

－GPF口：8只输入/输出口

－GPG口：16只输入/输出口

－GPH口：11只输入/输出口

每只输入/输出口（IO口）都可以很方便地通过软件来修改它们的配置。由于这些IO口的配置过程都完全类似，因此我们接下来主要以GPF口做为对象，来进行讲解。

S3C2410的每组IO口，均有着复用的功能。例如GPF口即可以做为输入口，或输出口，还可以定义为中断触发功能，所有的这些功能配置都是通过CPU的寄存器 GPFCON 来实现的。

GPFCON	Bit	Description
GPF7	[15:14]	00 = Input 01 = Output 10 = EINT7 11 = Reserved
GPF6	[13:12]	00 = Input 01 = Output 10 = EINT6 11 = Reserved
GPF5	[11:10]	00 = Input 01 = Output 10 = EINT5 11 = Reserved
GPF4	[9:8]	00 = Input 01 = Output 10 = EINT4 11 = Reserved
GPF3	[7:6]	00 = Input 01 = Output 10 = EINT3 11 = Reserved
GPF2	[5:4]	00 = Input 01 = Output 10 = EINT2 11 = Reserved
GPF1	[3:2]	00 = Input 01 = Output 10 = EINT1 11 = Reserved
GPF0	[1:0]	00 = Input 01 = Output 10 = EINT0 11 = Reserved

GPFCON 的第9、8位元用来配置GPF4（GPF口的第4只管脚）的功能，如果为00，配置为输入口；为01，配置为输出口；为10，则配置为EINT4中断触发口。在我们这个LED的驱动实验中，应该单独把GPF[7:4]设为输出口。但是从GPF口输出的电平具体是高电平还是低电平，完全是由 GPFDAT 寄存器中的资料来指定的。

GPFDAT	Bit	Description
GPF[7:0]	[7:0]	When the port is configured as input port. data from external sources can be read to the corresponding pin. When the port is configured as output port. data written in this register can be sent to the corresponding pin. When the port is configured as functional pin. undefined value will be read.

GPFDAT 是GPF口的资料寄存器。它的第0~7位分别对应GPF0~7的资料。当GPF为输出口时，对GPFDAT中的相应位写“1”，即可输出高电平；反之输出低电平。当GPF口为输入口时，从GPF口读到的资料即为GPF口相应管脚的外部电平状态。

当GPF口做为输入口时，还可以设置内部上拉电阻。

GPFUP	Bit	Description
GPF[7:0]	[7:0]	0: The pull-up function attached to to the corresponding port pin is enabled. 1: The pull-up function is disabled.

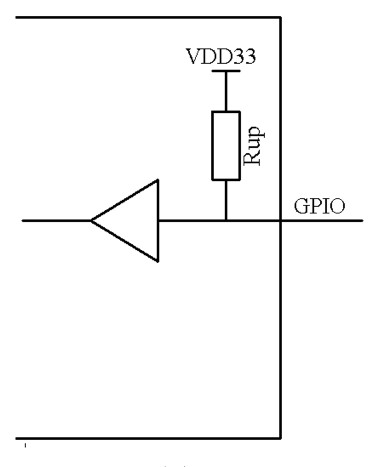


图4-3

图4-3是CPU内部的上拉电阻示意图，当GPF口设为输入口并使能上拉电阻时，即使GPIO的外部悬空，仍能保证CPU内部的GPIO为固定的状态（Rup仅仅起到微弱上拉的功能，当外部输入信号变化的时候，GPIO的状态可以跟随变化）。

第五章 异步串口通讯

数据通信的基本方式可分为并行通信与串行通信两种：

并行通信：是指利用多条数据传输线将一个资料的各位同时传送。它的特点是传输速度快，适用于短距离通信，但要求通讯速率较高的应用场合。

串行通信：是指利用一条传输线将资料一位位地顺序传送。特点是通信线路简单，利用简单的线缆就可实现通信，降低成本，适用于远距离通信，但传输速度慢的应用场合。

一、异步通信及其协议

异步通信以一个字符为传输单位，通信中两个字符间的时间间隔是不固定的，然而在同一个字符中的两个相邻位代码间的时间间隔是固定的。

通信协议（通信规程）：是指通信双方约定的一些规则。在使用异步串口传送一个字符的信息时，对资料格式有如下约定：规定有空闲位、起始位、资料位、奇偶校验位、停止位。

异步通讯的时序，如图5-1。

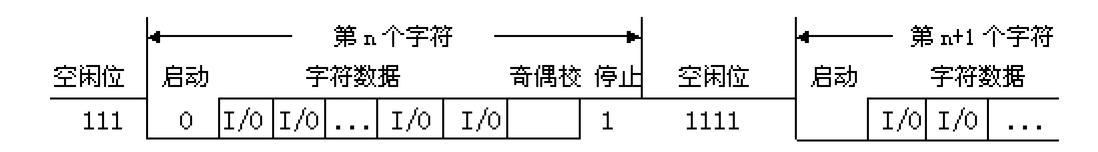


图 5-1

其中各位的意义如下：

起始位：先发出一个逻辑”0”信号，表示传输字符的开始。

资料位：紧接着起始位之后。资料位的个数可以是 4、5、6、7、8 等，构成一个字符。通常采用 ASCII 码。从最低位开始传送，靠时钟定位。

奇偶校验位：资料位加上这一位后，使得“1”的位数应为偶数(偶校验)或奇数(奇校验)，以此来校验资料传送的正确性。

停止位：它是一个字符数据的结束标志。可以是 1 位、1.5 位、2 位的高电平。

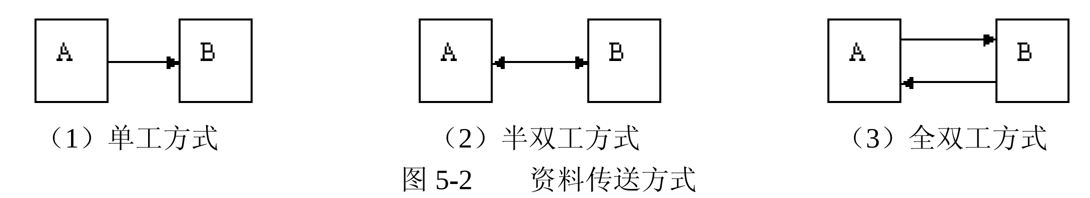
空闲位：处于逻辑“1”状态，表示当前线路上没有资料传送。

波特率：是衡量资料传送速率的指针。表示每秒钟传送的二进制位数。例如资料传送速率为 120 字符/秒，而每一个字符为 10 位，则其传送的波特率为 10×120＝1200 字符/秒＝1200 波特。

注：异步通信是按字符传输的，接收设备在收到起始信号之后只要在一个字符的传输时间内能和发送设备保持同步就能正确接收。下一个字符起始位的到来又使同步重新校准（依靠检测起始位来实现发送与接收方的时钟自同步的）。

二、资料传送方式

根据资料传送方向的不同有以下三种方式。如图 5-2 所示。



（1）单工方式　　（2）半双工方式　　（3）全双工方式

图 5-2　　资料传送方式

1、单工方式

资料始终是从 A 设备发向 B 设备。

2、 半双工方式

资料能从 A 设备传送到 B 设备，也能从 B 设备传送到 A 设备。在任何时候资料都不能同时在两个方向上传送，即每次只能有一个设备发送，另一个设备接收。但是通讯双方依照一定的通讯协议来轮流地进行发送和接收。

3、 全双工方式

允许通信双方同时进行发送和接收。这时，A 设备在发送的同时也可以接收，B 设备亦同。全双工方式相当于把两个方向相反的单工方式组合在一起，因此它需要两条数据传输线。在计算机串行通讯中主要使用半双工和全双工方式。

三、信号传输方式

1、基带传输方式

在传输线路上直接传输不加调制的二进制信号，如图所示。它要求传送线的频带较宽，传输的数字信号是矩形波。

基带传输方式仅适宜于近距离和速度较低的通信。

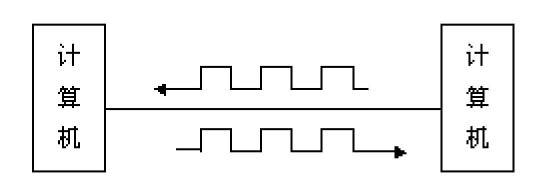


图 5-3

2、频带传输方式

传输经过调制的模拟信号

在长距离通信时，发送方要用调制器把数字信号转换成模拟信号，接收方则用解调器将接收到的模拟信号再转换成数字信号，这就是信号的调制解调。

实现调制和解调任务的装置称为调制解调器(MODEM)。采用频带传输时，通信双方各接一个调制解调器，将数字信号寄载在模拟信号(载波)上加以传输。因此，这种传输方式也称为载波传输方式。这时的通信线路可以是电话交换网，也可以是专用线。

常用的调制方式有三种：

调幅、调频和调相，分别如下图所示。

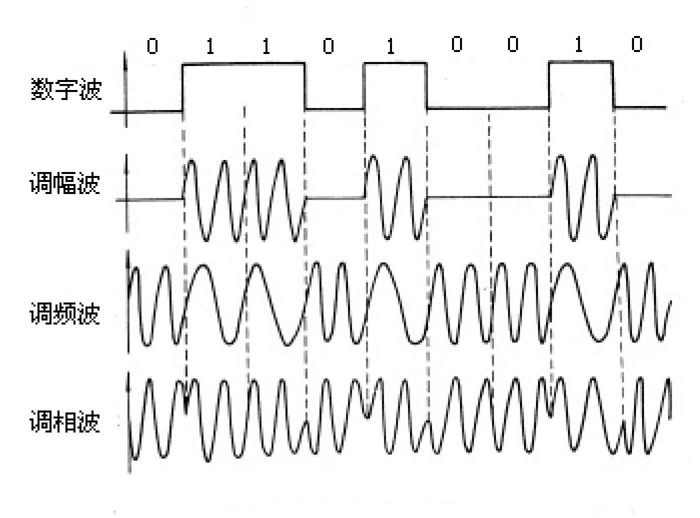


图 5-4

四、串行接口标准

串行接口标准：指的是计算机或终端(资料终端设备 DTE)的串行接口电路与调制解调器 MODEM 等(数据通信设备 DCE)之间的连接标准。

RS-232C 标准

RS-232C 是一种标准接口，D 型插座，采用 25 芯引脚或 9 芯引脚的连接器，如图 5-5 所示。

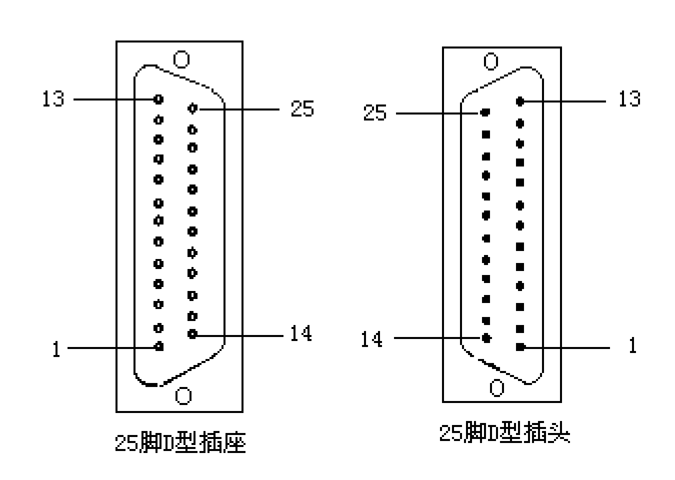
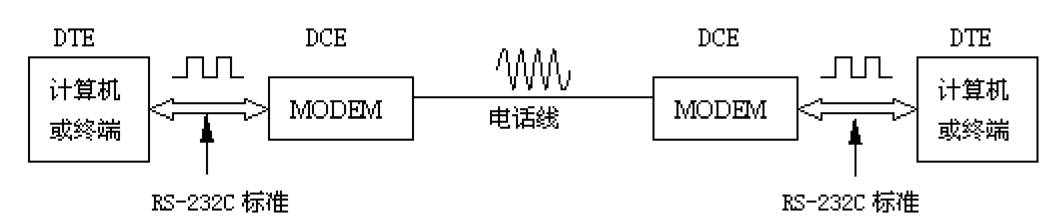


图 5-5

微型计算机之间的串行通信就是按照 RS-232C 标准设计的接口电路实现的。如果使用一根电话线进行通信，那么计算机和 MODEM 之间的联机就是根据 RS-232C 标准连接的。其连接及通信原理如图 5-6 所示



RS232 信号定义

图 5-6

RS-232C 标准规定接口有 25 根联机。只有以下 9 个信号经常使用.

引脚和功能分别如下：

1. TXD（第 2 脚）：发送资料线，输出。发送资料到 MODEM。
2. RXD（第 3 脚）：接收资料线，输入。接收资料到计算机或终端。
3. $\overline{\text{RTS}}$（第 4 脚）：请求发送，输出。计算机通过此引脚通知 MODEM，要求发送资料。
4. $\overline{\text{CTS}}$（第 5 脚）：允许发送，输入。发出$\overline{\text{CTS}}$作为对$\overline{\text{RTS}}$的回答，计算机才可以进行发送资料。
5. $\overline{\text{DSR}}$（第 6 脚）：资料装置就绪(即 MODEM 准备好)，输入。表示调制解调器可以使用，该信号有时直接接到电源上，这样当设备连通时即有效。
6. CD（第 8 脚）：载波检测(接收线信号测定器)，输入。表示 MODEM 已与电话线路连接好。
7. 如果通信线路是交换电话的一部分，则至少还需如下两个信号：

8. RI（第 22 脚）：振铃指示，输入。MODEM 若接到交换台送来的振铃呼叫信号，就发出该信号来通知计算机或终端。
9. $\overline{\text{DTR}}$（第 20 脚）：资料终端就绪，输出。计算机收到 RI 信号以后，就发出 $\overline{\text{DTR}}$ 信号到 MODEM 作为回答，以控制它的转换设备，建立通信链路。
10. GND（第 7 脚）：信号地

逻辑电平

RS-232C 标准采用 EIA 电平，规定：

“1”的逻辑电平在-3V~-15v 之间

“0”的逻辑电平在+3V~+15V 之间。

由于 EIA 电平与 TTL 电平完全不同，必须进行相应的电平转换，MCl488 完成 TTL 电平到 EIA 电平的转换，MCl489 完成 EIA 电平到 ITL 电平的转换。还有 MAX232 可以同时完成 TTL->EIA 和 EIA->TTL 的电平转换。

除了 RS-232C 标准以外，还有一些其它的通用的异步串行接口标准，如：

RS-423A 标准

为了克服 RS-232C 的缺点，提高传送速率，增加通信距离，又考虑到与 RS-232C 的兼容性，美国电子工业协会在 1987 年提出了 RS-423A 标准。该标准的主要优点是在接收端采用了差分输入。而差分输入对共模干扰信号有较高的抑制作用，这样就提高了通信的可靠性。RS-423A 用-6v 表示逻辑“1”，用+6v 表示逻辑“0”，可以直接与 RS-232C 相接。采用 RS-423A 标准以获得比 RS-232C 更佳的通信效果。图 5-7 是 RS423A 的连接示意图。

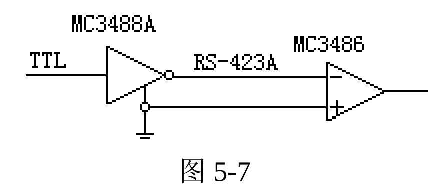


图 5-7

RS-422A 标准

RS-422A 总线采用平衡输出的发送器，差分输入的接收器。如图 5-8 所示。

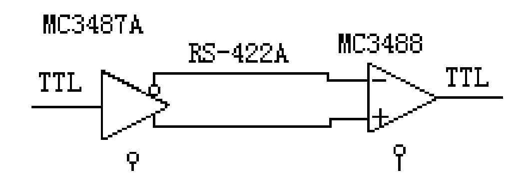


图 5-8

RS-422A 的输出信号线间的电压为±2v，接收器的识别电压为±0.2v。共模范围±25v。在高速传送信号时，应该考虑到通信线路的阻抗匹配，一般在接收端加终端电阻以吸收掉反射波。电阻网络也应该是平衡的，如图 5-9 所示。

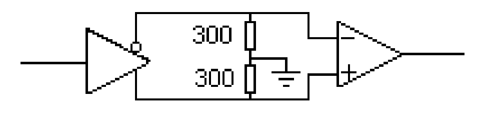


图 5-9 为 RS-422A 平衡输出差分输示意图

图 5-9

RS-485 标准

RS-485 适用于收发双方共享一对线进行通信，也适用于多个点之间共享一对线路进行总线方式联网，但通信只能是半双工的，线路如图 5-10 所示。

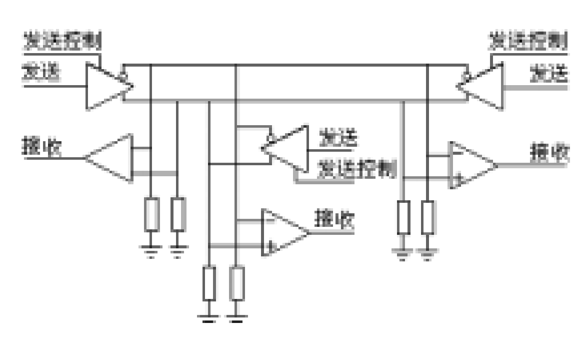


图 5-10

典型的 RS232 到 RS422/485 转换芯片有：MAX481/483/485/487/488/489/490/491，SN75175/176/184 等等，它们均只需单一+5v 电源供电即可工作（芯片内部采用电荷泵方式升压）。具体使用方法可查阅有关技术手册。

五、S3C2410 内置的 UART 控制器

S3C2410内部具有3个独立的UART控制器，每个控制器都可以工作在Interrupt（中断）模式或DMA（直接内存访问）模式，也就是说UART控制器可以CPU与UART控制器传送资料的时候产生中断或DMA请求。并且每个UART均具有16字节的FIFO（先入先出寄存器），支持的最高波特率可达到230.4Kbps

图5-11是S3C2410内部UART控制器的结构图

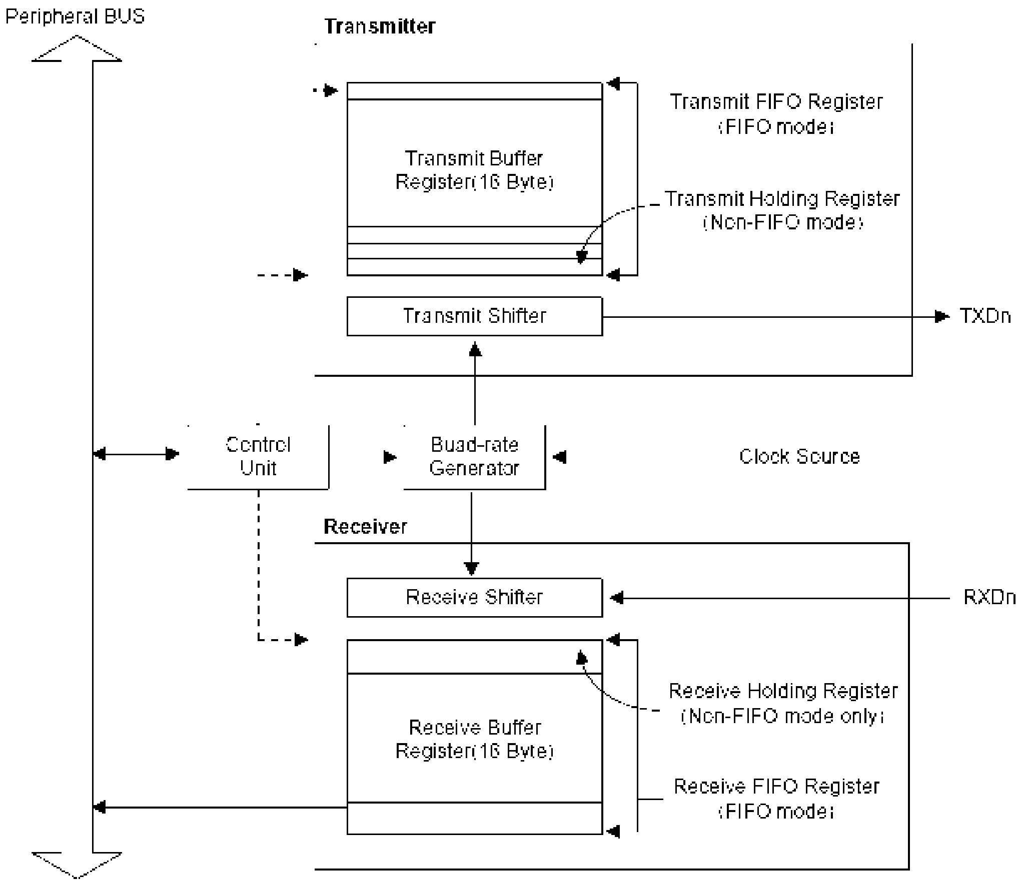


图5-11

UART的操作

UART的操作分为以下几个部分，分别是：资料发送、资料接收、产生中断、产生波特率、Loopback模式、红外模式以及自动流控模式。

资料发送

发送的资料帧格式是可以编程设置的。它包含了起始位、5~8个资料位、可选的奇偶校验位以及1~2位停止位。这些都是通过UART的控制寄存器 ULCONn 来设置的。

资料接收

同发送一样，接收的资料帧格式也是可以进行编程设置的。此外，还具备了检测溢出出错、奇偶校验出错、帧出错等出错检测，并且每种错误都可以置相应的错误标志。

自动流控模式

S3C2410的UART0和UART1都可以通过各自的nRTS和nCTS信号来实现自动流控。

在自动流控（AFC）模式下nRTS取决于接收端的状态，而nCTS控制了发送断的操作。具体地说：只有当nCTS有效时（表明接收方的FIFO已经准备就绪来接收资料了），UART才会将FIFO中的资料发送出去。在UART接收资料之前，只要当接收FIFO有至少2-byte空余的时候，nRTS就会被置为有效。图5-12是UART 自动流控模式的连接方式

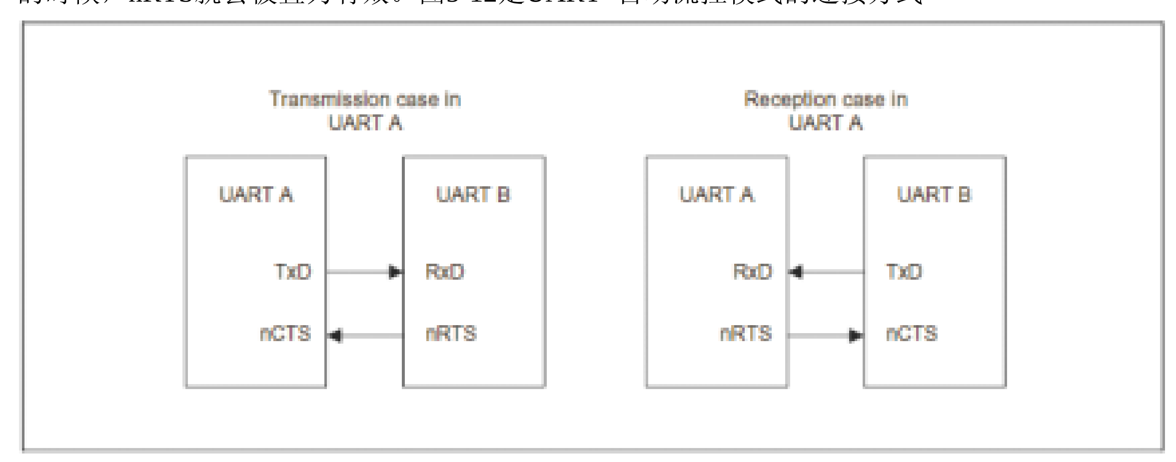


图5-12

中断/DMA请求产生

S3C2410的每个UART都有7种状态，分别是：溢出覆盖（Overrun）错误、奇偶校验错误、帧出错、断线错误、接收就绪、发送缓冲空闲、发送移位器空闲。它们在UART状态寄存器 UTRSTATn / UERSTATn 中有相应的标志位。

波特率发生器

每个UART控制器都有各自的波特率发生器来产生发送和接收资料所用的序列时钟，波特率发生器的时钟源可以CPU内部的系统时钟，也可以从CPU的 UCLK 管脚由外部取得时钟信号，并且可以通过 UCONn 选择各自的时钟源。

波特率产生的具体计算方法如下：

当选择CPU内部时钟时：

UBRDIVn=(int)(PCLK/(bps*16))-1,bps为所需要的波特率值，PCLK为CPU内部外设总线（APB）的工作时钟。

当需要得到更精确的波特率时，可以选择由 UCLK 引入的外部时钟来生成。

UBRDIVn=(int)(UCLK/(bps*16))-1

LoopBack操作模式

S3C2410 CPU的UART提供了一种测试模式，也就是这里所说的LoopBack模式。在设计

系统的具体应用时，为了判断通讯故障是由于外部的数据链路上的问题，还是CPU内驱动程序或CPU本身的问题，这就需要采用LoopBack模式来进行测试。在LoopBack模式中，资料发送端TXD在UART内部就从逻辑上与接收端RXD连在一起，并可以来验证资料的收发是否正常。

UART控制寄存器

下面将针对UART的各个控制寄存器逐一进行讲解，以期对UART的操作和设置能有更进一步的了解。

ULCONn（**UART Line Control Register**）见图5-13

ULCONn	Bit	Description	Initial State
Reserved	[7]		0
Infra-Red Mode	[6]	Determine whether or not to use the Infra-Red mode. 0 = Normal mode operation 1 = Infra-Red Tx/Rx mode	0
Parity Mode	[5:3]	Specify the type of parity generation and checking during UART transmit and receive operation. 0xx = No parity 100 = Odd parity 101 = Even parity 110 = Parity forced/checked as 1 111 = Parity forced/checked as 0	000
Number of Stop Bit	[2]	Specify how many stop bits are to be used for end-of-frame signal. 0 = One stop bit per frame 1 = Two stop bit per frame	0
Word Length	[1:0]	Indicate the number of data bits to be transmitted or received per frame. 00 = 5-bits 01 = 6-bits 10 = 7-bits 11 = 8-bits	00

图5-13

Word Length ：资料位长度

Number of Stop Bit：停止位数

Parity Mode：奇偶校验位类型

Infra-Red Mode ：UART/红外模式选择（当以UART模式工作时，需设为“0”）

UCONn（**UART Control Register**）见图5-14

Receive Mode：选择接收模式。如果是采用DMA模式的话，还需要指定说使用的DMA信道。

Transmit Mode：同上。

Send Break Signal：选择是否在传1帧资料中途发送Break信号。

Loopback Mode：选择是否将UART置于Loopback测试模式。

Rx Error Status Interrupt Enable：选择是否使能当发生接收异常时，是否产生接收错误中断。

Rx Time Out Enable：是否使能接收超时中断。

Rx Interrupt Type：选择接收中断类型。

选择0：Pulse（脉冲式/边沿式中断。非FIFO模式时，一旦接收缓冲区中有资料，即产生一个中断；为FIFO模式时，一旦当FIFO中的资料达到一定的触发水平后，即产生一个中断）

选择1：Level（电平模式中断。非FIFO模式时，只要接收缓冲区中有资料，即产生中断；为FIFO模式时，只要FIFO中的资料达到触发水平后，即产生中断）

Tx Interrupt Type：类同于Rx Interrupt Type

Clock Selection：选择UART波特率发生器的时钟源。

UCONn	Bit	Description	Initial State
Clock Selection	[10]	Select PCLK or UCLK for the UART baud rate. 0=PCLK : UBRDIVn = (int)(PCLK / (bps x 16)) -1 1=UCLK(@GPH8) : UBRDIVn = (int)(UCLK / (bps x 16)) -1	0
Tx Interrupt Type	[9]	Interrupt request type. 0 = Pulse (Interrupt is requested as soon as the Tx buffer becomes empty in Non-FIFO mode or reaches Tx FIFO Trigger Level in FIFO mode.) 1 = Level (Interrupt is requested while Tx buffer is empty in Non-FIFO mode or reaches Tx FIFO Trigger Level in FIFO mode.)	0
Rx Interrupt Type	[8]	Interrupt request type. 0 = Pulse (Interrupt is requested the instant Rx buffer receives the data in Non-FIFO mode or reaches Rx FIFO Trigger Level in FIFO mode.) 1 = Level (Interrupt is requested while Rx buffer is receiving data in Non-FIFO mode or reaches Rx FIFO Trigger Level in FIFO mode.)	0
Rx Time Out Enable	[7]	Enable/Disable Rx time out interrupt when UART FIFO is enabled. The interrupt is a receive interrupt. 0 = Disable 1 = Enable	0
Rx Error Status Interrupt Enable	[6]	Enable the UART to generate an interrupt upon an exception, such as a break, frame error, parity error, or overrun error during a receive operation. 0 = Do not generate receive error status interrupt. 1 = Generate receive error status interrupt.	0
Loopback Mode	[5]	Setting loopback bit to 1 causes the UART to enter the loopback mode. This mode is provided for test purposes only. 0 = Normal operation 1 = Loopback mode	0
Send Break Signal	[4]	Setting this bit causes the UART to send a break during 1 frame time. This bit is automatically cleared after sending the break signal. 0 = Normal transmit 1 = Send break signal	0
Transmit Mode	[3:2]	Determine which function is currently able to write Tx data to the UART transmit buffer register. 00 = Disable 01 = Interrupt request or polling mode 10 = DMA0 request (Only for UART0), DMA3 request (Only for UART2) 11 = DMA1 request (Only for UART1)	00
Receive Mode	[1:0]	Determine which function is currently able to read data from UART receive buffer register. 00 = Disable 01 = Interrupt request or polling mode 10 = DMA0 request (Only for UART0),	00

图5-14

UFCONn（**UART FIFO Conrtol Register**）见图5-15

FIFO Enable：FIFO使能选择。

Rx FIFO Reset：选择当复位接收FIFO时是否自动清除FIFO中的内容。

Tx FIFO Reset：选择当复位发送FIFO时是否自动清除FIFO中的内容。

Rx FIFO Trigger Level：选择接收FIFO的触发水平。

Tx FIFO Trigger Level：选择发送FIFO的触发水平。

UFCONn	Bit	Description	Initial State
Tx FIFO Trigger Level	[7:6]	Determine the trigger level of transmit FIFO. 00 = Empty 01 = 4-byte 10 = 8-byte 11 = 12-byte	00
Rx FIFO Trigger Level	[5:4]	Determine the trigger level of receive FIFO. 00 = 4-byte 01 = 8-byte 10 = 12-byte 11 = 16-byte	00
Reserved	[3]		0
Tx FIFO Reset	[2]	Auto-cleared after resetting FIFO 0 = Normal 1= Tx FIFO reset	0
Rx FIFO Reset	[1]	Auto-cleared after resetting FIFO 0 = Normal 1= Rx FIFO reset	0
FIFO Enable	[0]	0 = Disable 1 = Enable	0

图5-15

UMCONn（**UART Modem Control Register**）见图5-16

Request to Send：如果在AFC模式下，该位元将由UART控制器自动设置；否则的话就必须由用户的软件来控制。

Auto Flow Control：选择是否使能自动流控（AFC）。

UMCONn	Bit	Description	Initial State
Reserved	[7:5]	These bits must be 0's	00
Auto Flow Control (AFC)	[4]	0 = Disable 1 = Enable	0
Reserved	[3:1]	These bits must be 0's	00
Request to Send	[0]	If AFC bit is enabled, this value will be ignored. In this case the S3C2410X will control nRTS automatically. If AFC bit is disabled, nRTS must be controlled by software. 0 = 'H' level (Inactivate nRTS) 1 = 'L' level (Activate nRTS)	0

图5-16

UTRSTATn（**UART TX/RX Status Register**）见图5-17

Receive buffer data ready:当接收缓冲寄存器从UART接收端口接收到有效资料时将自动置“1”。反之为“0”则表示缓冲器中没有资料。

Transmit buffer empty：当发送缓冲寄存器中为空，自动置“1”；反之表明缓冲器中正有资料等待发送。

Transmitter empty：当发送缓冲器中已经没有有效资料时，自动置“1”；反之表明尚有资料未发送。

UTRSTATn	Bit	Description	Initial State
Transmitter empty	[2]	Set to 1 automatically when the transmit buffer register has no valid data to transmit and the transmit shift register is empty. 0 = Not empty 1 = Transmitter (transmit buffer & shifter register) empty	1
Transmit buffer empty	[1]	Set to 1 automatically when transmit buffer register is empty. 0 =The buffer register is not empty 1 = Empty (In Non-FIFO mode, Interrupt or DMA is requested. In FIFO mode, Interrupt or DMA is requested, when Tx FIFO Trigger Level is set to 00 (Empty)) If the UART uses the FIFO, users should check Tx FIFO Count bits and Tx FIFO Full bit in the UFSTAT register instead of this bit.	1
Receive buffer data ready	[0]	Set to 1 automatically whenever receive buffer register contains valid data, received over the RXDn port. 0 = Empty 1 = The buffer register has a received data (In Non-FIFO mode, Interrupt or DMA is requested) If the UART uses the FIFO, users should check Rx FIFO Count bits and Rx FIFO Full bit in the UFSTAT register instead of this bit.	0

图5-17

UERSTATn（**UART Error Status Register**）见图5-18

Overrun Error：为“1”，表明发生Overrun错误。

Frame Error：为“1”。表明发生Frame（帧）错误。

UERSTATn	Bit	Description	Initial State
Reserved	[3]	0 = No frame error during receive 1 = Frame error (Interrupt is requested.)	0
Frame Error	[2]	Set to 1 automatically whenever a frame error occurs during receive operation. 0 = No frame error during receive 1 = Frame error (Interrupt is requested.)	0
Reserved	[1]	0 = No frame error during receive 1 = Frame error (Interrupt is requested.)	0
Overrun Error	[0]	Set to 1 automatically whenever an overrun error occurs during receive operation. 0 = No overrun error during receive 1 = Overrun error (Interrupt is requested.)	0

图5-18

UFSTATn：（**UART FIFO Status Register**）见图5-19

Rx FIFO Count:接收FIFO中当前存放的字节数。

Tx FIFO Count:发送FIFO中当前存放的字节数。

Rx FIFO Full:为“1“表明接收FIFO已满。

Tx FIFO Full:为“1“表明发送FIFO已满。

UFSTATn	Bit	Description	Initial State
Reserved	[15:10]		0
Tx FIFO Full	[9]	Set to 1 automatically whenever transmit FIFO is full during transmit operation 0 = 0-byte ≤ Tx FIFO data ≤ 15-byte 1 = Full	0
Rx FIFO Full	[8]	Set to 1 automatically whenever receive FIFO is full during receive operation 0 = 0-byte ≤ Rx FIFO data ≤ 15-byte 1 = Full	0
Tx FIFO Count	[7:4]	Number of data in Tx FIFO	0
Rx FIFO Count	[3:0]	Number of data in Rx FIFO	0

图5-19

UMSTATn ：（**UART FIFO Status Register**）见图5-20

Clear to Send ：为“0”表示CTS无效；为“1”表示CTS有效。

Delta CTS ：指示自从上次CPU访问该位后，nCTS的状态有无发生改变。

为“0”则说明不曾改变；反之表明nCTS信号已经变化了。

UMSTAT0	Bit	Description	Initial State
Reserved	[3]		0
Delta CTS	[2]	Indicate that the nCTS input to the S3C2410X has changed state since the last time it was read by CPU. (Refer to Figure 11-8.) 0 = Has not changed 1 = Has changed	0
Reserved	[1]		0
Clear to Send	[0]	0 = CTS signal is not activated (nCTS pin is high.) 1 = CTS signal is activated (nCTS pin is low.)	0

图5-20

UTXHn 和 **URXHn** 分别是**UART**发送和接收资料寄存器

这两个寄存器存放着发送和接收的资料，当然只有一个字节 8 位资料。需要注意的是在发生溢出错误的时候，接收的资料必须要被读出来，否则会引发下次溢出错误

UBRDIVn ：（**UART Baud Rate Divisor Register**）见图5-21

UBRDIVn	Bit	Description	Initial State
UBRDIV	[15:0]	Baud rate division value UBRDIVn >0	–

图5-21

关于UART波特率的计算方法，在前面的内容中已经有详细的阐述，此处不做多余说明。

小结：读写状态寄存器UTRSTAT 以及错误状态寄存UERSTAT，可以反映芯片目前的读写状态以及错误类型。FIFO 状态寄存器UFSTAT 和MODEM 状态寄存器UMSTAT，通过前
者可以读出目前FIFO 是否满以及其中的字节数；通过后者可以读出目前MODEM 的CTS 状态。

第六章 触摸屏及模数转换

一、触摸屏的几个概念

所谓触摸屏，从市场概念来讲，就是一种人人都会使用的计算机输入设备，或者说是人人都会使用的与计算机沟通的设备。不用学习，人人都会使用，是触摸屏最大的魔力，这一点无论是键盘还是鼠标，都无法与其相比。

从技术原理角度讲，触摸屏是一套透明的绝对寻址系统，首先它必须保证是透明的，因此它必须通过材料科技来解决透明问题，像数字化仪、写字板、电梯开关，它们都不是触摸屏；其次它是绝对坐标，手指摸哪就是哪，不需要第二个动作，不像鼠标，是相对定位的一套系统，我们可以注意到，触摸屏软件都不需要游标，有游标反倒影响用户的注意力，因为游标是给相对定位的设备用的，相对定位的设备要移动到一个地方首先要知道现在在何处，往哪个方向去，每时每刻还需要不停的给用户反馈当前的位置才不致于出现偏差。这些对采取绝对坐标定位的触摸屏来说都不需要；再其次就是能检测手指的触摸动作并且判断手指位置，各类触摸屏技术就是围绕“检测手指触摸”而八仙过海各显神通的。

1、触摸屏的第一个指针：**光学特性**。它直接影响到触摸屏的视觉效果。但是触摸屏是多层的复合薄膜，光学特性上包括四个方面：透明度、色彩失真度、反光性和清晰度。彩色世界包含了可见光波段中的各种波长色，在没有完全解决透明材料科技之前，或者说还没有低成本的很好解决透明材料科技之前，多层复合薄膜的触摸屏在各波长下的透光性还不能达到理想的一致状态，下面是一个示意图（图 6-1）：

图 6-1

由于透光性与波长曲线图的存在，通过触摸屏看到的图像不可避免的与原图像产生了色彩失真，静态的图像感觉还只是色彩的失真，动态的多媒体图像感觉就不是很舒服了，色彩失真度也就是图中的最大色彩失真度自然是越小越好。平常所说的透明度也只能是图中的平均透明度，当然是越高越好。

反光性，主要是指由于镜面反射造成图像上重叠身后的光影，例如人影、窗户、灯光等。反光是触摸屏带来的负面效果，越小越好，它影响用户的浏览速度，严重时甚至无法辨认图像字符，反光性强的触摸屏使用环境受到限制，现场的灯光布置也被迫需要调整。大多数存在反光问题的触摸屏都提供另外一种经过表面处理的型号：磨砂面触摸屏，也叫防眩型，价格略高一些，防眩型反光性明显下降，适用于采光非常充足的大厅或展览场所，不过，防眩型的透光性和清晰度也随之有较大幅度的下降。

清晰度，有些触摸屏加装之后，字迹模糊，图像细节模糊，整个屏幕显得模模糊糊，看不太清楚，这就是清晰度太差。清晰度的问题主要是多层薄膜结构的触摸屏，由于薄膜层之间光反复反射折射而造成的，此外防眩型触摸屏由于表面磨砂也会造成清晰度下降。清晰度

不好，眼睛容易疲劳，对眼睛也有一定伤害，选购触摸屏时要注意判别。

2、触摸屏的第二个特性：**稳定性**。 触摸屏是绝对坐标系统，要选哪就直接点那，与鼠标这类相对定位系统的本质区别是一次到位的直观性。绝对坐标系的特点是每一次定位坐标与上一次定位坐标没有关系，触摸屏在物理上是一套独立的坐标定位系统，每次触摸的资料通过校准资料转为屏幕上的坐标，这样，就要求触摸屏这套坐标不管在什么情况下，同一点的输出资料是稳定的，如果不稳定，那么这触摸屏就不能保证绝对坐标定位，点不准，这就是触摸屏最怕的问题：漂移。技术原理上凡是不能保证同一点触摸每一次采样资料相同的触摸屏都免不了漂移这个问题，目前有漂移现象的只有电容触摸屏。

3、触摸屏的第三个特性：**相应性**。检测触摸并定位， 各种触摸屏技术都是依靠各自的传感器来工作的，甚至有的触摸屏本身就是一套传感器。各自的定位原理和各自所用的传感器决定了触摸屏的反应速度、可靠性、稳定性和寿命。触摸屏的传感器方式还决定了触摸屏如何识别多点触摸的问题，也就是超过一点的同时触摸怎么办？有人触摸时接着旁边又有人触摸怎么办？这是触摸屏使用过程中经常出现的问题，我认为最理想的方式是：超过一点的同时触摸谁也不判断，一直等到多点触摸移走，有人触摸接着又有人触摸应该是分先后都判断，当然是技术上可能的话。

红外触摸屏靠多对红外发射和接收对管来工作，红外对管性能和寿命都比较可靠，任何阻挡光线的物体都可用来作触摸物，不过红外触摸屏使用传感器数目将近 100 对， 并且共享外围电路，这就要求传感器不仅本身性能好，还要求将近 100 对的红外二极管“光-电阻特性”和“结电容”都保持一致。实际应用中，万一有哪一对出现故障，可以在上电自检过程中发现并在此后加以忽略，靠邻近的红外线代替，由于每一对红外线只“监管”约 6mm 左右的窄带，而手指通常在 15mm 左右粗细，用户是察觉不到的。但如果生产过程没有对红外发射管进行老化测试，没有很好的质量管理体系，将近 100 对的传感器， 很快就不是一对两对“掉队”的问题了，总体寿命也就难以保证。下图（图 6-2）是红外触摸屏的示意图

图 6-2

电容触摸屏本身实际上是一套精密的漏电传感器，带手套的手不能触摸，由于使用电容方式，导致有漂移现象，在下节电容触摸屏有详细的介绍。超声波触摸屏有表面声波触摸屏和体波声波触摸屏，利用的都是电－声压电换能器作传感器，接收传感器和发射传感器所用

的压晶体管不是一种型号，在制造时的掺杂材料略有不同，发射换能器功率大，接收换能器更加灵敏。压电换能器的寿命长，工作稳定，正常工作可以保证 10 年不出问题。触摸屏安装后，换能器是隐藏起来的，但是在运输和安装过程中需要小心谨慎，裸露的换能器晶体不能碰撞挤压。表面声波触摸屏有 X、Y 轴两对传感器，利用屏幕表面的声表面波来检测手指触摸，可以说，工作面是一层看不见、打不坏的声能，不怕暴力使用，最适合公共信息查询。

以上谈了一些触摸屏技术领域的概念，当然，只是是纯技术原理的一些探讨，评判一种触摸屏，光是技术原理还只是其中的一部分，触摸屏要应用到各个领域，还要抵受千触万摸，选用材料的耐用性如何，反应速度如何，价格能否承受，这些都是理性的评判一种触摸屏。

由于目前基于电阻技术的触摸屏由于价格低廉，亦可满足绝大多数，下面着重介绍一下电阻式触摸屏的基本原理：

电阻触摸屏的屏体部分是一块与显示器表面非常配合的多层复合薄膜，由一层玻璃或有机玻璃作为基层，表面涂有一层透明的导电层，上面再盖有一层外表面硬化处理、光滑防刮的塑料层，它的内表面也涂有一层透明导电层，在两层导电层之间有许多细小（小于千分之一英寸）的透明隔离点把它们隔开绝缘。如图 6-3

图 6-3 电阻触摸屏剖面结构

图 6-4

当手指触摸屏幕时，平常相互绝缘的两层导电层就在触摸点位置有了一个接触，因其中一面导电层接通 Y 轴方向的 5V 均匀电压场，使得侦测层的电压由零变为非零，控制器侦测到这个接通后，进行 A/D 转换，并将得到的电压值与 5V 相比即可得触摸点的 Y 轴坐标，同理得出 X 轴的坐标，这就是所有电阻技术触摸屏共同的最基本原理。

电阻类触摸屏的关键在于材料科技。常用的透明导电涂层材料有：

①ITO，氧化铟，弱导电体，特性是当厚度降到 1800 个埃（埃＝10-10 米）以下时会突然变得透明，透光率为 80%，再薄下去透光率反而下降，到 300 埃厚度时又上升到 80%。ITO 是所有电阻技术触摸屏及电容技术触摸屏都用到的主要材料，实际上电阻和电容技术触摸屏的工作面就是 ITO 涂层。

②镍金涂层，五线电阻触摸屏的外层导电层使用的是延展性好的镍金涂层材料，外导电层由于频繁触摸，使用延展性好的镍金材料目的是为了延长使用寿命，但是工艺成本较为高昂。镍金导电层虽然延展性好，但是只能作透明导体，不适合作为电阻触摸屏的工作面，因为它导电率高，而且金属不易做到厚度非常均匀，不宜作电压分布层，只能作为探层。

五线电阻触摸屏：

五线电阻技术触摸屏的基层把两个方向的电压场通过精密电阻网络都加在玻璃的导电工作面上，我们可以简单的理解为两个方向的电压场分时工作加在同一工作面上，而外层镍金导电层只仅仅用来当作纯导体，有触摸后分时检测内层 ITO 接触点 X 轴和 Y 轴电压值的方法测得触摸点的位置。 五线电阻触摸屏内层 ITO 需四条引线，外层只作导体仅仅一条，触摸屏得引出线共有 5 条。五线制电阻触摸屏的结构如图 6-5

图 6-5 五线制触摸屏的结构

四线电阻触摸屏的缺陷：

电阻触摸屏的 B 面要经常被触动，四线电阻触摸屏的 B 面采用 ITO，我们知道， ITO 是极薄的氧化金属，在使用过程中，很快就会产生细小的裂纹，而裂纹一旦产生，原流经该处的电流被迫绕裂纹而行，本该均匀分布的电压随之遭到破坏，触摸屏就有了损伤，表现为裂纹处点不准。

图 6-6 四线制触摸屏的裂纹导致分流

随着裂纹的加剧和增多，触摸屏慢慢就会失效，因此使用寿命不长是四线电阻触摸屏的主要问题。

五线电阻触摸屏的改进：

首先五线电阻触摸屏的 A 面是导电玻璃而不是导电涂覆层，导电玻璃的工艺使得 A 面的寿命得到极大的提高，并且可以提高透光率。

其次五线电阻触摸屏把工作面的任务都交给寿命长的 A 面，而 B 面只用来作为导体，并且采用了延展性好、电阻率低的镍金透明导电层，因此，B 面的寿命也极大的提高。

五线电阻触摸屏的另一个专有技术是通过精密的电阻网络来校正 A 面 的线性问题：由于工艺工程不可避免的有可能厚薄不均而造成电压场不均匀分布，精密电阻网络在工作时流过绝大部分电流，因此可以补偿工作面有可能的线性失真。

五线电阻触摸屏是目前最好的电阻技术触摸屏，最适合于军事、医疗领域使用。

但是四线电阻触摸屏由于价格低廉，在通用领域的运用，下面将结合 S3C2410 内置的触摸屏控制器来详细讲解整个触摸屏电路的工作及测量过程。

下图是四线电阻触摸屏测量时的等效电路（图 6-7）：

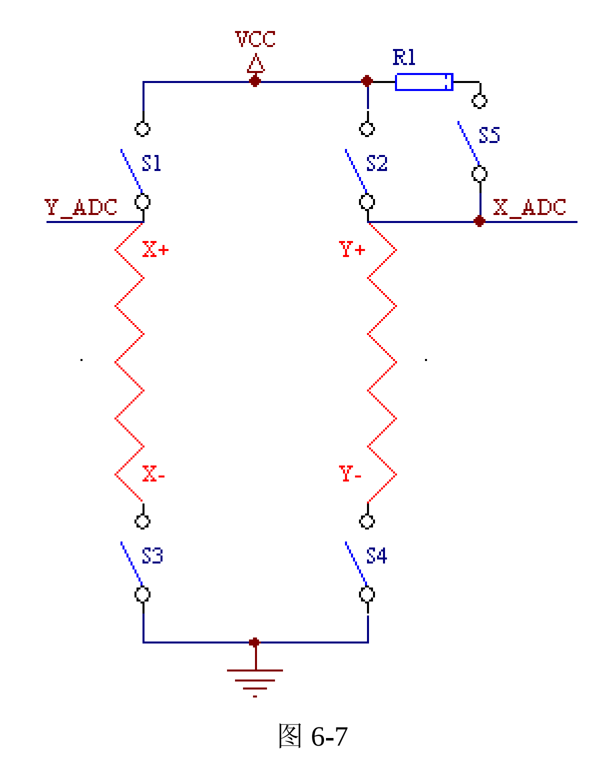


图 6-7

测量时，分为以下 3 个步骤：

（1） 起初，在触摸屏没有被按下的时候，触摸屏的 X 轴和 Y 轴不会接触在一起，此时这个电路处在“Pen Down Detect”状态。S1、S2、S4 断开，S3、S5 闭合。X+~X-的整个轴上的电压均为 0V（GND），Y-端悬空，Y+端由于有上拉电阻 R1 的存在而呈现高电平。当“Pen Down”后，X 轴和 Y 轴受挤压而接触导通后，Y 轴上的电压由于连通到 X 轴接地而变为低电平，此低电平可做为中断触发信号来通知 CPU 发生“Pen Down”事件。

（2） 当检测到“PenDown”事件后，CPU 立刻进入 X 轴坐标测量状态：S1、S3 闭合，S2、S4、S5 断开（Y+、Y-两断悬空）。由于 X 轴和 Y 轴在接触点按下而连通，因此 Y＋端的 X_ADC 可以认为是 X 轴的分压采样点（通过测量 X_ADC 的电压可以得到 X+到接触点，以及 X-到接触点的比例），从而计算出 X 轴的坐标

（3） 采样完 X 轴的坐标后，S1、S3、S5 断开，S2、S4 闭合，同样原理，我们可以进一步得到 Y 轴的坐标。

二、S3C2410 模数转换器（ADC）及触摸屏控制器

S3C2410内置1个8信道的10bit模数转换器（ADC），该ADC能以500KSPS的采样资料将外部的模拟信号转换为10bit分辨率的数字量。同时ADC部分能与CPU的触摸屏控制器协同工作，完成对触摸屏绝对地址的测量。

特性：

－分辨率：10bit

－相信误差：＋/- 2LSB

－最大转换速率：500KSPS

－模拟量输入范围：0～3.3V

－分步 X/Y坐标测量模式

－自动X/Y坐标测量模式

－中断等待模式

下图是 ADC 及触摸屏控制器部分的逻辑示意图（图 6-8）

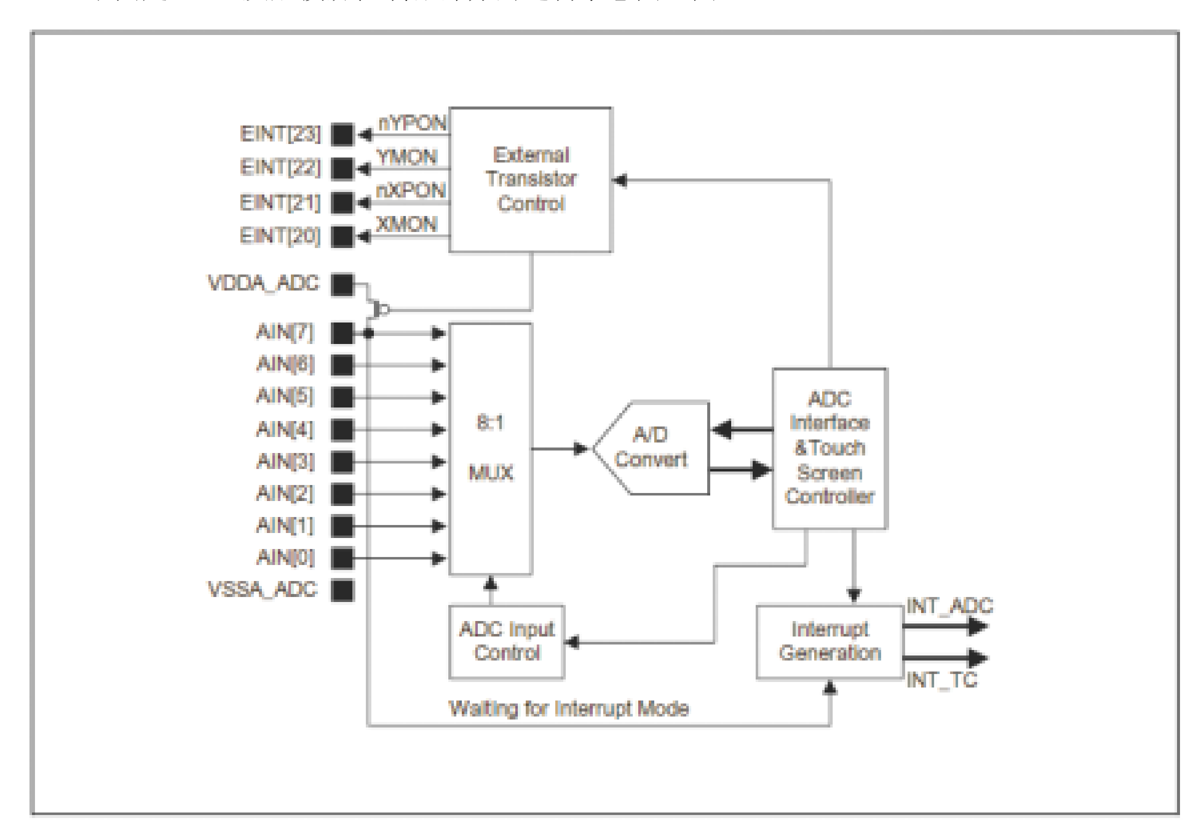
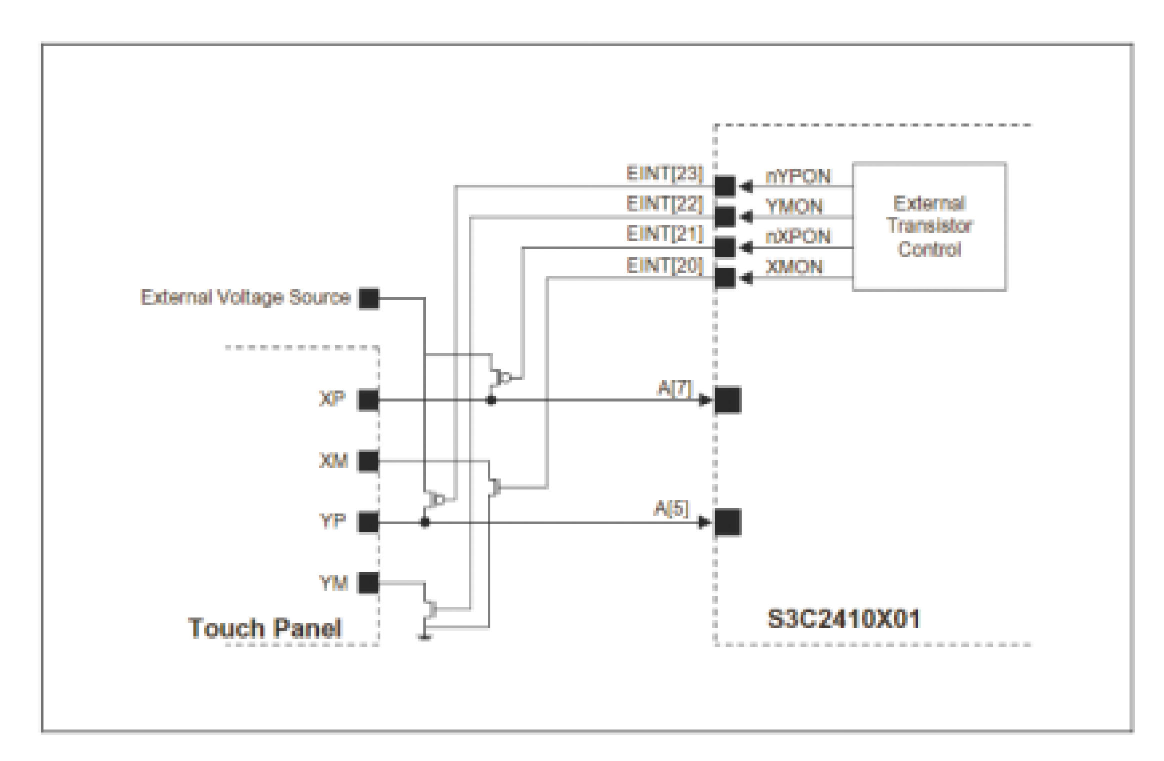


图6-8

随后的图是在S3C2410的ADC以及触摸屏控制器的基础上外接触摸屏的示意图，以及外部电路的实际原理图。需要补充说明的是，图中Q1、Q2为P沟道MOS管，开门电压为1.8V；Q3、Q4为N沟道MOS管，开门电压为2.7V。运用学过的电子电路的知识，我们知道当MOS管导通后（栅极电压达到开门电压之后），MOS管的源-漏极之间可以认为是直通的（导通电阻为毫欧级），即可以把MOS管认为是图4-7中真正的“开关”。AVDD是外部模拟参考源，一般接3.3V电源，XP、XM和YP、YM分别是触摸屏的4条引线，各自对应X轴和Y轴电阻。



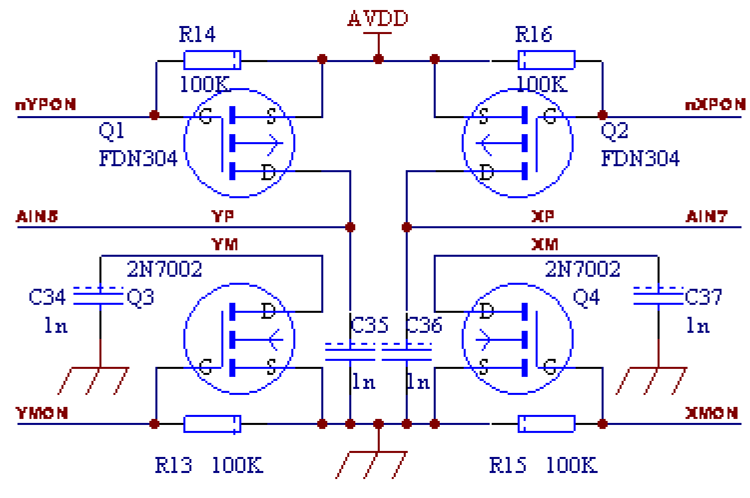


图6-8

ADC及触摸屏控制器的工作模式：

1、 ADC普通转换模式（Normal Converson Mode）

普通转换模式（AUTO_PST=0,XY_PST=0）是用来进行一般的ADC转换之用的，例如通过ADC测量电池电压等等。

2、 独立X/Y轴坐标转换模式（Separate X/Y Position Conversion Mode）

独立X/Y轴坐标转换模式其实包含了X轴模式和Y轴模式2种模式。

首先进行X轴的坐标转换（AUTO_PST=0，XY_PST=1），X轴的转换资料会写到ADCDAT0寄存器的XPDAT中，等待转换完成后，触摸屏控制器会产生相应的中断。

然后进行Y轴的坐标转换（AUTO_PST=0，XY_PST=2），Y轴的转换资料会写到ADCDAT1寄存器的YPDAT中，等待转换完成后，触摸屏控制器会产生相应的中断。

3、 自动X/Y轴坐标转换模式（Auto X/Y Position Conversion Mode）

自动X/Y轴坐标转换模式（AUTO_PST=1，XY_PST=0）将会自动地进行X轴和Y轴的转换操作，随后产生相应的中断。

4、 中断等待模式（Wait for InterruptMode）

在系统等待“Pen Down”，即触摸屏按下的时候，其实是处于中断等待模式。一旦被按下，实时产生“INT_TC”中断信号。每次发生此中断都，X轴和Y轴坐标转换资料都可以从相应的资料寄存器中读出。

5、 闲置模式（Standby Mode）

在该模式下转换资料寄存器中的值都被保留为上次转换时的资料。

ADC及触摸屏控制器的寄存器详解

ADCCON：**ADC控制寄存器**（见图6-9）

ENABLE_START ：

置1：启动ADC转换

置0：无操作

RESR_START ：

置1：允许读操作启动ADC转换

置0：禁止读操作启动ADC转换

STDBM：

置1：将ADC置为闲置状态（模式）

置0：将ADC置为正常操作状态

SEL_MUX：选择需要进行转换的ADC信道

PRSCVL：ADC转换时钟预分频参数

PRSCEN：ADC转换时钟使能

ECFLG：ADC转换完成标志位（只读）

为1：ADC转换结束

为0：ADC转换进行中

ADCCON	Bit	Description	Initial State
ECFLG	[15]	End of conversion flag (read only). 0 = A/D conversion in process 1 = End of A/D conversion	0
PRSCEN	[14]	A/D converter prescaler enable. 0 = Disable 1 = Enable	0
PRSCVL	[13:6]	A/D converter prescaler value. Data value: 1 ~ 255 Note that division factor is (N+1) when the prescaler value is N.	0xFF
SEL_MUX	[5:3]	Analog input channel select. 000 = AIN 0 001 = AIN 1 010 = AIN 2 011 = AIN 3 100 = AIN 4 101 = AIN 5 110 = AIN 6 111 = AIN 7 (XP)	0
STDBM	[2]	Standby mode select. 0 = Normal operation mode 1 = Standby mode	1
READ_ START	[1]	A/D conversion start by read. 0 = Disable start by read operation 1 = Enable start by read operation	0
ENABLE_ START	[0]	A/D conversion starts by setting this bit. If READ_START is enabled, this value is not valid. 0 = No operation 1 = A/D conversion starts and this bit is cleared after the start-up.	0

图6-9

ADCTSC：**触摸屏控制寄存器**（见图6-10）

XY_PST：对X/Y轴手动测量模式进行选择

AUTO_PST：X/Y轴的自动转换模式使能位元

PULL_UP ：XP端的上拉电阻使能位

XP_SEN：设置nXPON输出状态

XM_SEN ：设置XMON输出状态

YP_SEN：设置nYPON输出状态

YM_SEN ：设置YMON输出状态

ADCTSC	Bit	Description	Initial State
Reserved	[8]	This bit should be zero.	0
YM_SEN	[7]	Select output value of YMON. 0 = YMON output is 0 (YM = Hi-Z). 1 = YMON output is 1 (YM = GND).	0
YP_SEN	[6]	Select output value of nYPON. 0 = nYPON output is 0 (YP = External voltage). 1 = nYPON output is 1 (YP is connected with AIN[5]).	1
XM_SEN	[5]	Select output value of XMON. 0 = XMON output is 0 (XM = Hi-Z). 1 = XMON output is 1 (XM = GND).	0
XP_SEN	[4]	Select output value of nXPON. 0 = nXPON output is 0 (XP = External voltage). 1 = nXPON output is 1 (XP is connected with AIN[7]).	1
PULL_UP	[3]	Pull-up switch enable. 0 = XP pull-up enable 1 = XP pull-up disable	1
AUTO_PST	[2]	Automatically sequencing conversion of X-position and Y-position 0 = Normal ADC conversion 1 = Auto (Sequential) X/Y Position Conversion Mode	0
XY_PST	[1:0]	Manual measurement of X-position or Y-position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for Interrupt Mode	0

图6-10

ADCDLY：**ADC转换周期等待定时器**（见图6-11）

ADCDLY	Bit	Description	Initial State
DELAY	[15:0]	1) Normal Conversion Mode, Separate X/Y Position Conversion Mode, and Auto (Sequential) X/Y Position Conversion Mode. → X/Y Position Conversion Delay Value. 2) Waiting for Interrupt Mode. When Stylus down occurs in Waiting for Interrupt Mode, this register generates Interrupt signal (INT_TC) at intervals of several ms for Auto X/Y Position conversion. **NOTE:** Do not use Zero value (0x0000)	00ff

图6-11

ADCDAT0：**ADC资料寄存器0**（见图6-12）

XPDATA ：X轴转换资料寄存器

XY_PST：选择X/Y轴自动转换模式

AUTO_PST：X/Y轴自动转换使能位

UPDOWN ：选择中断等待模式的类型

为0：按下产生中断

为1：释放产生中断

ADCDAT0	Bit	Description	Initial State
UPDOWN	[15]	Up or down state of Stylus at Waiting for Interrupt Mode. 0 = Stylus down state 1 = Stylus up state	-
AUTO_PST	[14]	Automatic sequencing conversion of X-position and Y-position. 0 = Normal ADC conversion 1 = Sequencing measurement of X-position, Y-position	-
XY_PST	[13:12]	Manual measurement of X-position or Y-position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for Interrupt Mode	-
Reserved	[11:10]	Reserved	
XPDATA (Normal ADC)	[9:0]	X-position conversion data value. (include Normal ADC conversion data value) Data value: 0 ~ 3FF	-

图6-12

ADCDAT1: **ADC资料寄存器1**（见图6-13）

定义类同于ADCDAT0。

ADCDAT1	Bit	Description	Initial State
UPDOWN	[15]	Up or down state of Stylus at Waiting for Interrupt Mode. 0 = Stylus down state 1 = Stylus up state	-
AUTO_PST	[14]	Automatically sequencing conversion of X-position and Y-position. 0 = Normal ADC conversion 1 = Sequencing measurement of X-position, Y-position	-
XY_PST	[13:12]	Manual measurement of X-position or Y-position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for Interrupt Mode	-
Reserved	[11:10]	Reserved	
YPDATA	[9:0]	Y-position conversion data value Data value: 0 ~ 3FF	-

图6-13

第七章 IIC 总线驱动及 IIC EEPROM 的操作

一、IIC 总线的现状

IIC 总线是 Philips 公司开发的一种简单、双向、二线制、同步串行总线。它只需两根线（ SDA 和 SCL ）即可在连接于总线上的器件之间传送信息。该总线是高性能串行总线，具备多主机系统所需要的裁决和高低速设备同步等功能，简化了硬件设计，解决了很多在设计数字元控制电路中遇到的接口问题。目前 IIC 总线的应用已经极为广泛，实际上已经成为一个国际标准，在超过 100 种不同的 IC 上实现而且得到超过 50 家公司的许可。

以下是 IIC 总线的一些特征：

（1） 只要求两条总线线路一条串行资料线 SDA 一条串行时钟线 SCL

（2） 每个连接到总线的器件都可以通过唯一的地址和一直存在的简单的主机从机关系软件设定地址主机可以作为主机发送器或主机接收器

（3） 它是一个真正的多主机总线如果两个或更多主机同时初始化数据传输可以通过冲突检测和仲裁防止资料被破坏

（4） 串行的 8 位双向数据传输位元速率在标准模式下可达 100kbit/s 快速模式下可达 400kbit/s

（5） 片上的滤波器可以滤去总线资料线上的毛刺波保证资料完整

（6） 连接到相同总线的 IC 数量只受到总线的最大电容 400pF 限制

二、IIC 总线的概念

IIC 总线支持任何一种类型的 IC，无论是 NMOS、CMOS 还是 TTL。无论是 CPU、LCD驱动、内存或键盘接口，每个器件都有一个唯一的地址识别码，而且都可以作为一个发送器或接收器。很明显 LCD驱动只是一个接收器，而内存则既可以接收又可以发送资料。除了发送器和接收器外，器件在执行数据传输时也可以被看作是主机或从机(见图7-1)。主机是初始化总线的数据传输并产生允许传输的时钟信号的器件，此时任何被寻址的器件都被认为是从机。

术语	描述
发送器	发送数据到总线的器件
接收器	从总线接收数据的器件
主机	初始化发送、产生时钟信号和终止发送的器件
从机	被主机寻址的器件
多主机	同时有多于一个主机尝试控制总线，但不破坏报文
仲裁	是一个在有多个主机同时尝试控制总线，但只允许其中一个控制总线并使报文不被破坏的过程
同步	两个或多个器件同步时钟信号的过程

图7-1

SDA 和 SCL 都是双向线路，都通过一个上拉电阻连接到正的电源电压。当总线空闲时这两条线路都是高电平。连接到总线的器件输出级必须是漏极开路或集电极开路才能实现线与的功能。IIC 总线上资料的传输速率在标准模式下可达 100kbit/s 在快速模式下可达 400kbit/s。

SDA 线上的资料必须在时钟的高电平周期保持稳定，资料线的高或低电平状态只有在 SCL 线的时钟信号是低电平时才能改变（在 IIC 总线上产生时钟信号是由主机器件来实现的。

当在总线上传输资料时，每个主机产生自己的时钟信号）。如图7-2。

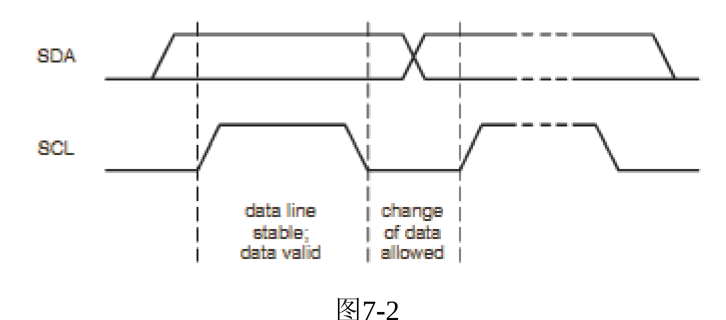


图7-2

在 IIC 协议中，定义了一系列的时序做为通讯协议。其中一种情况是在 SCL 线是高电平时，SDA 线从高电平向低电平切换，这个情况表示起始条件。当 SCL 是高电平时，SDA 线由低电平向高电平切换，表示停止条件。如图7-3

起始和停止条件一般由主机产生，总线在起始条件后被认为处于忙的状态，在停止条件的某段时间后总线被认为再次处于空闲状态。如果产生重复起始 Sr 条件，而不产生停止条件总线会一直处于忙的状态，此时的起始条件 S 和重复起始 Sr 条件在功能上是一样的。

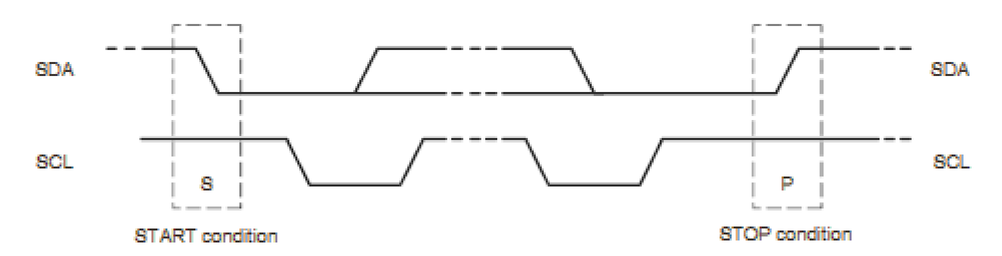


图7-3

发送到 SDA 线上的每个字节必须为 8 位，每次传输可以发送的字节数量不受限制。每个字节后必须跟一个响应位。首先传输的是资料的最高位 MSB（见图7-4），如果从机要完成一些其它功能后才能接收或发送下一个完整的资料字节，可以使时钟线 SCL 保持低电平迫使主机进入等待状态。当从机准备好接收下一个数据字节并释放时钟线 SCL 后，数据传输继续。

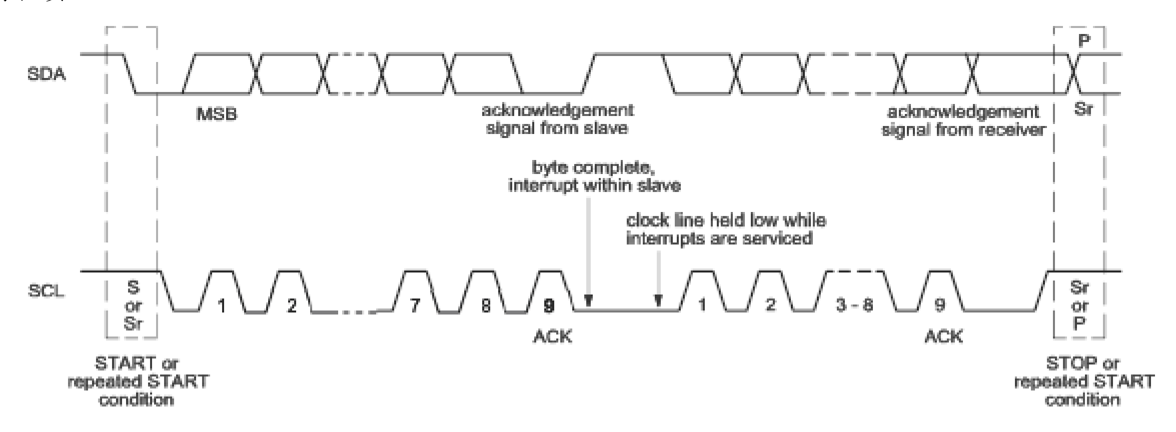


图7-4

数据传输必须带回应，相关的响应时钟脉冲由主机设备产。生在响应的时钟脉冲期间发送器释放 SDA 线。

在响应的时钟脉冲期间接收器必须将 SDA 线拉低使它在这个时钟脉冲的高电平期间保持稳定的低电平。

资料的传输遵循图7-5 所示的格式。在起始条件 S 后发送了一个从机地址这个地址(共有7 位），紧接着的第 8 位是资料方向位 R/W（ 0 表示发送写，1 表示请求资料读）数据传输一般由主机产生的停止位 P 终止，但是如果主机仍希望在总线上通讯它可以产生重复起始条件 Sr。

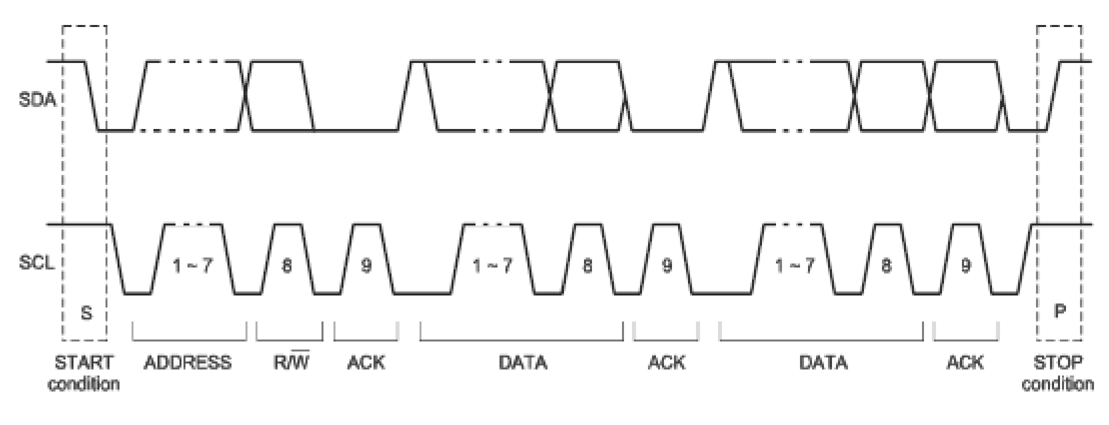


图7-5

三、基于 IIC 总线的 EEPROM：AT24C04-SC27

AT24C04-SC27 是 Atmel 公司所产之基于 IIC 总线的 EEPROM，它具有以下特性：

（1） 低工作电压（2.7V~5.5V）
（2） 内部存储结构为512×8（4Kbyte）
（3） IIC 总线接口（ 100KHz 总线速度）
（4） 硬件写保护端口
（5） 16Byte页面写入模式
（6） 写入周期10ms
（7） 高可靠性
－擦除/写入周期：100万次
－资料保存期：100年

四、S3C2410 的 IIC 总线控制器

S3C2410 内部的 IIC 总线控制器支持多主 IIC 总线。在多主 IIC 总线模式下，多个S3C2410（或其它 IIC 主机）可以通过 IIC 总线对从机初始化、终止或收发资料。下图是S3C2410 内部的 IIC 控制器的逻辑框图（图7-6）

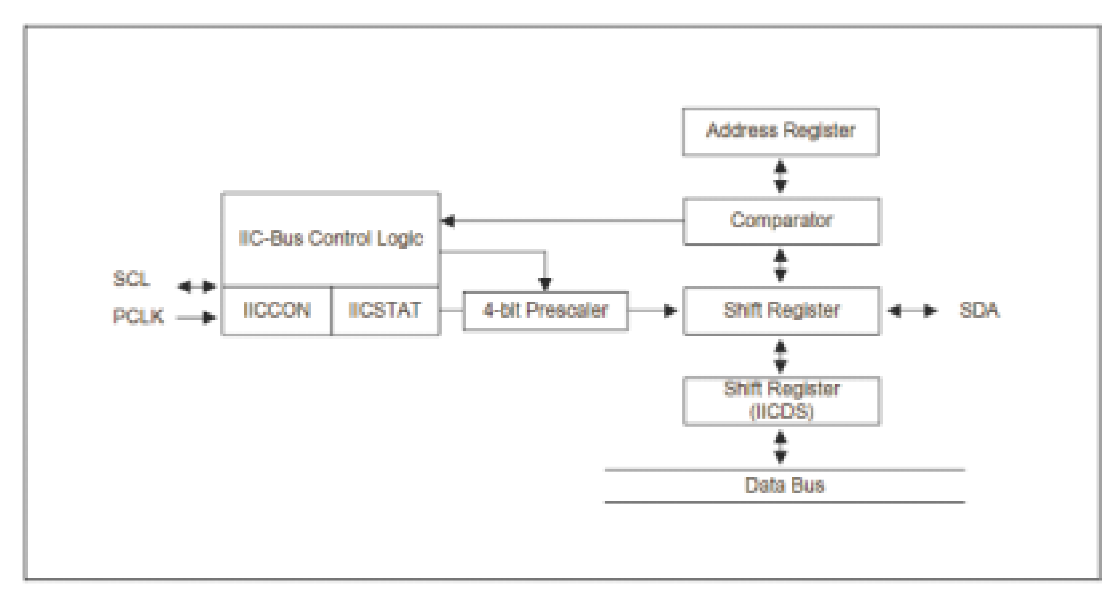


图7-6

S3C2410 的 IIC 总线控制器共有 4 种操作模式，分别是：

（1） 主机发送模式（图7-7是工作流程）

图7-7

（2） 主机接收模式（图7-8是工作流程）

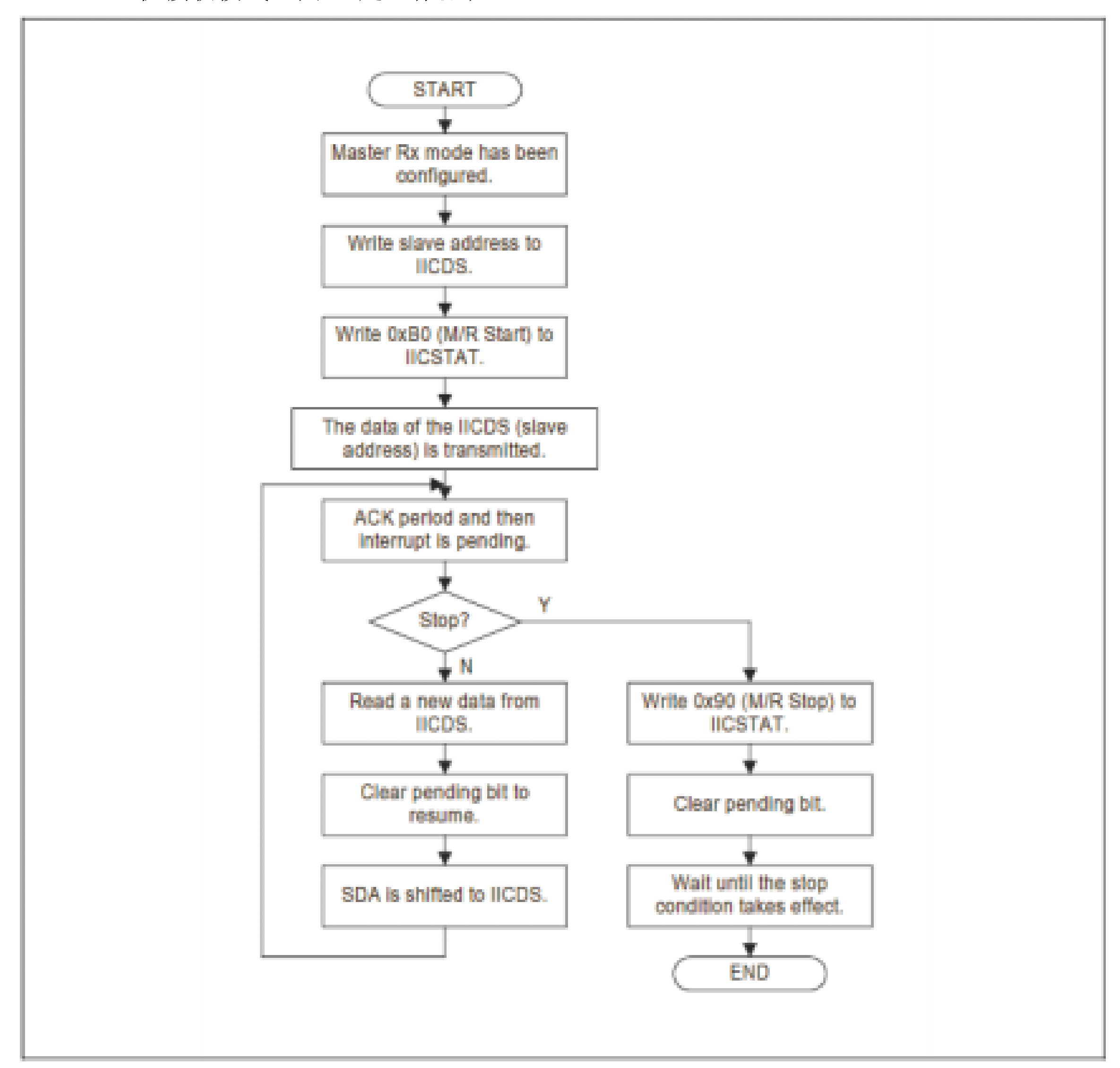


图7-8

（3） 从机发送模式（图7-9是工作流程）

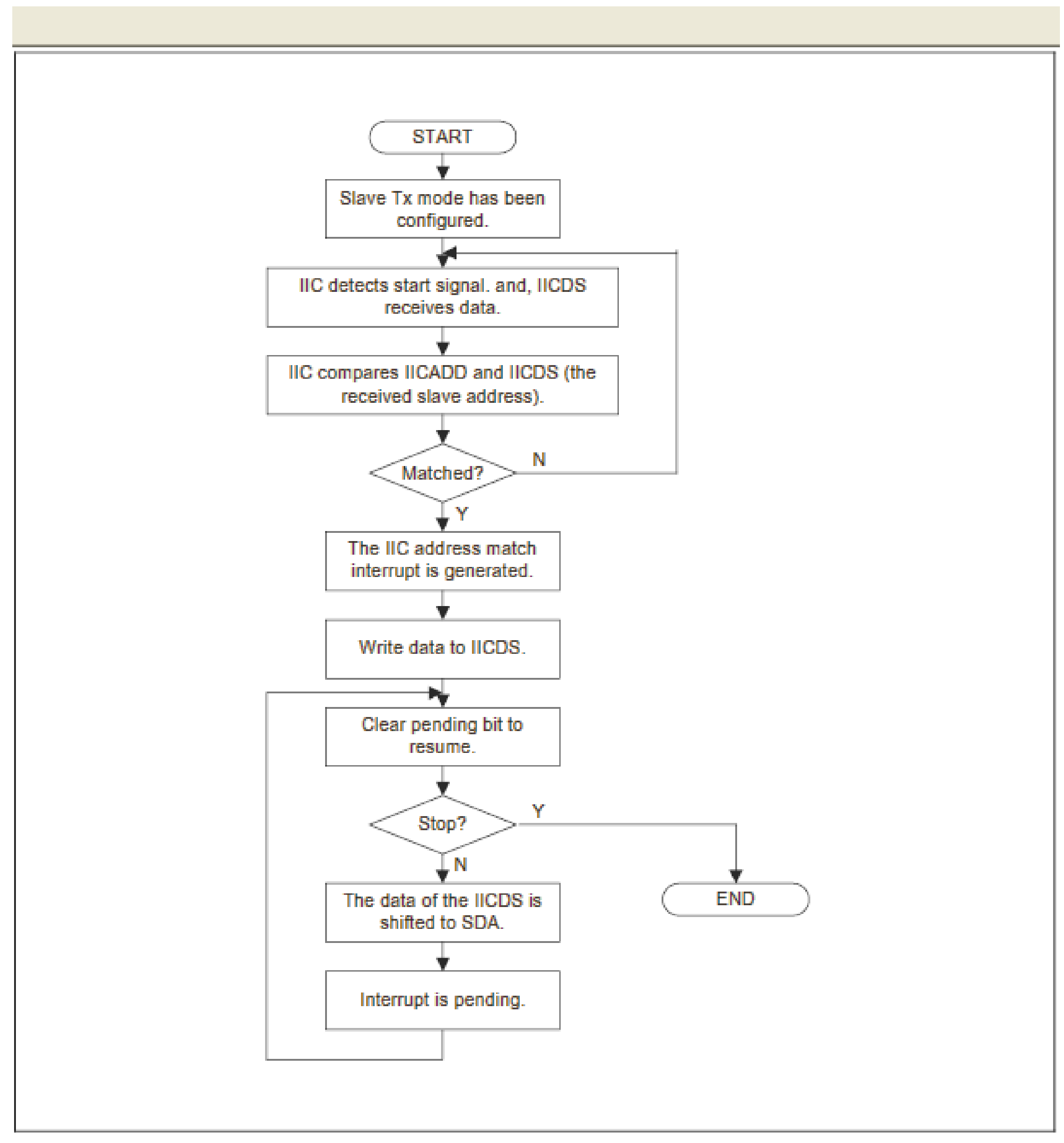


图7-9

（4） 从机接收模式（图7-10是工作流程）

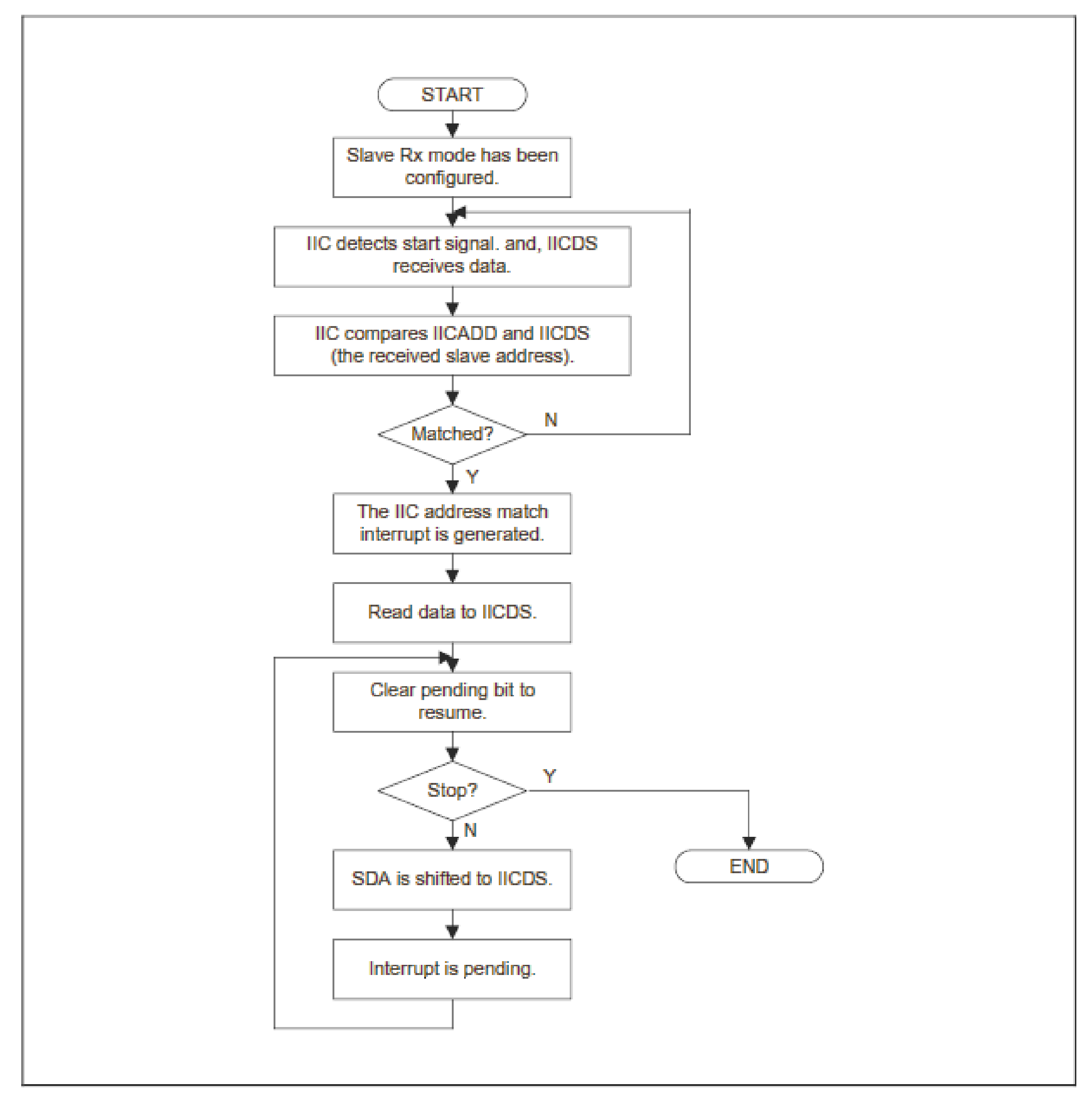


图7-10

五、S3C2410 IIC 控制寄存器详解

IICCON：**IIC 多主总线控制寄存器**（见图7-11）

Transmit Clock Value ：IIC 总线发送时钟预分频值。IIC 的发送时钟是根据下面的计算方法得到的：发送时钟 Tx Clock = IICCLK /（IICCON[3:0]＋1）

Interrupt Pending Flag：IIC 总线中断悬挂标志位

读该位资料：

为0：没有中断被挂起

为1：已有中断挂起

写该位资料：

置0：清除中断悬挂标志位

置1：无效

Tx/Rx Interrupt：发送/接收中断使能位

Tx Clock Source Selection：IIC 发送时钟源选择位

置0：选择 IIC 的发送时钟为PCLK的1/16（PCLK为CPU的外设总线时钟）

置1：选择 IIC 的发送时钟为PCLK的1/512

响应位位产生使能位：

置0：不产生响应位

置1：产生响应位

IICCON	Bit	Description	Initial State
Acknowledge generation (note 1)	[7]	IIC-bus acknowledge enable bit. 0 = Disable, 1 = Enable In Tx mode, the IICSDA is free in the ack time. In Rx mode, the IICSDA is L in the ack time.	0
Tx clock source selection	[6]	Source clock of IIC-bus transmit clock prescaler selection bit. 0 = IICCLK = f_{PCLK} /16 1 = IICCLK = f_{PCLK} /512	0
Tx/Rx Interrupt (note 5)	[5]	IIC-Bus Tx/Rx interrupt enable/disable bit. 0 = Disable, 1 = Enable	0
Interrupt pending flag (note 2), (note 3)	[4]	IIC-bus Tx/Rx interrupt pending flag. This bit cannot be written to 1. When this bit is read as 1, the IICSCL is tied to L and the IIC is stopped. To resume the operation, clear this bit as 0. 0 = 1) No interrupt pending (when read). 2) Clear pending condition & Resume the operation (when write). 1 = 1) Interrupt is pending (when read) 2) N/A (when write)	0
Transmit clock value (note 4)	[3:0]	IIC-Bus transmit clock prescaler. IIC-Bus transmit clock frequency is determined by this 4-bit prescaler value, according to the following formula: Tx clock = IICCLK/(IICCON[3:0]+1).	Undefined

图7-11

IICSTAT：**多主 IIC 总线控制器状态寄存器**（见图7-12）

Last-Received bit Status flag：IIC 总线最后接收位状态

为 0：最后接收到的是“0”（即收到响应位）

为 1：没有收到响应位

Address zero status flasg：IIC 总线“0”地址状态标志位

为 0：当检测到起始位或停止位后，清除 IIC 从设备地址

为 1：接收设备的从地址为 0x00

Address-as-slave status flasg：

为 0：当检测到起始位或停止位后，清除S3C2410做为 IIC 从设备的地址

为 1：接收到的从设备地址符合 IICADD中的值

Arbitration status flag：IIC 总线仲裁状态位

为 0：IIC 总线仲裁成功

为 1：IIC 总线仲裁失败

Serial output：IIC 总线资料输出使能

Busy signal status：IIC 总线“忙”信号状态位

为 0：IIC 总线空闲（已产生停止位）

为 1：IIC 总线忙（已产生起始位）

Mode Selection： IIC 总线 主/从 收/发模式选择

IICSTAT	Bit	Description	Initial State
Mode selection	[7:6]	IIC-bus master/slave Tx/Rx mode select bits. 00: Slave receive mode 01: Slave transmit mode 10: Master receive mode 11: Master transmit mode	00
Busy signal status / START STOP condition	[5]	IIC-Bus busy signal status bit. 0 = read) Not busy (when read) write) STOP signal generation 1 = read) Busy (when read) write) START signal generation. The data in IICDS will be transferred automatically just after the start signal.	0
Serial output	[4]	IIC-bus data output enable/disable bit. 0 = Disable Rx/Tx, 1 = Enable Rx/Tx	0
Arbitration status flag	[3]	IIC-bus arbitration procedure status flag bit. 0 = Bus arbitration successful 1 = Bus arbitration failed during serial I/O	0
Address-as-slave status flag	[2]	IIC-bus address-as-slave status flag bit. 0 = Cleared when START/STOP condition was detected 1 = Received slave address matches the address value in the IICADD	0
Address zero status flag	[1]	IIC-bus address zero status flag bit. 0 = Cleared when START/STOP condition was detected. 1 = Received slave address is 00000000b.	0
Last-received bit status flag	[0]	IIC-bus last-received bit status flag bit. 0 = Last-received bit is 0 (ACK was received). 1 = Last-received bit is 1 (ACK was not received).	0

图7-12

IICADD：**IIC** 总线从设备地址（见图7-13）

IICADD	Bit	Description	Initial State
Slave address	[7:0]	7-bit slave address, latched from the IIC-bus. When serial output enable = 0 in the IICSTAT, IICADD is write-enabled. The IICADD value can be read any time, regardless of the current serial output enable bit (IICSTAT) setting. Slave address = [7:1] Not mapped = [0]	XXXXXXXX

图7-13

IICDS：**IIC** 总线资料寄存器（见图7-14）

IICDS	Bit	Description	Initial State
Data shift	[7:0]	8-bit data shift register for IIC-bus Tx/Rx operation. When serial output enable = 1 in the IICSTAT, IICDS is write-enabled. The IICDS value can be read any time, regardless of the current serial output enable bit (IICSTAT) setting.	XXXXXXXX

图7-14

第八章 IIS 数字音频总线驱动及音频 DAC

一、概述

音频系统设计包括软件设计和硬件设计两方面。在硬件上使用了基于 IIS 总线的音频系统体系结构。IIS(Inter-IC Sound bus)又称 I2S，是 Philips 与 SONY 共同提出的串行数字音频总线协议。并从 80年代的第1代 CD 唱机开始就得到了广泛的应用，并沿用至今。目前已经成为了全球性的工业标准，与之类似的还有近几年来才开始得到应用的 AC97 标准。

下图是 S3C2410 IIS 控制器的内部逻辑示意图：

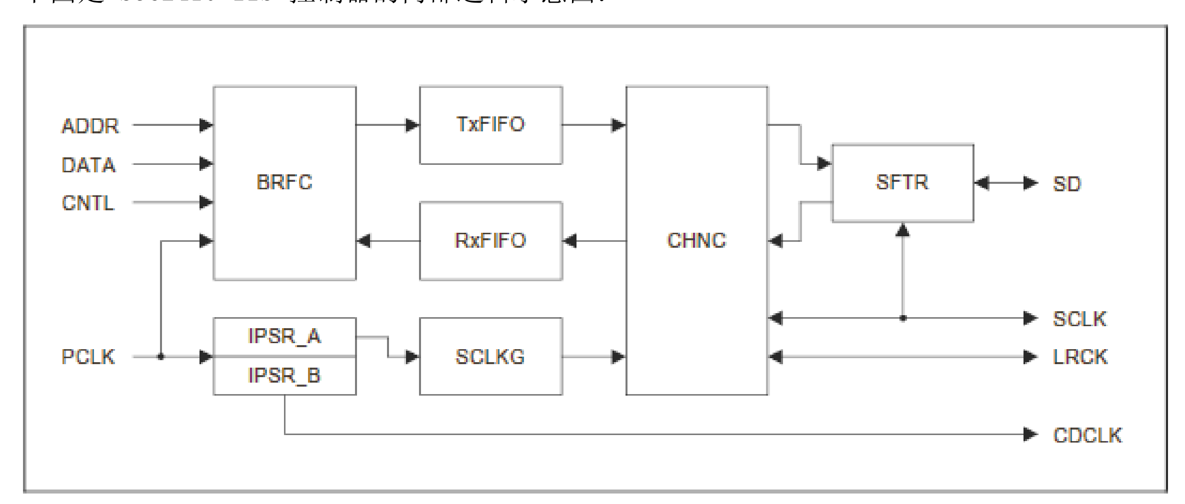


图8-1

二、数字音频接口格式

标准 IIS-BUS 格式

IIS 总线由 4 条信号组成，分别是 串行数据输入（IISSDI）、串行数据输出（IISSDO）、左右声道选择（IISLRCK），位时钟（IISCLK），此外音频 DAC 还需要工作时钟信号 MCLK。

串行资料是以 MSB-first（高位先出）的方式逐一移位元元发送，之所以这样是因为发送端发送的资料长度也许与接收端所能接收的资料长度不同，同时发送端也并不知道接收端能否处理接收到的资料。

IISLRCK不但是采样频率，可用来指示当前传送的资料是左声道资料，还是右声道资料的，并同时可用来控制 IIS 从设备（例如DAC）锁存当前接收到的资料，并清空接收下一个资料的输入缓冲区。

MSB（Left）Justified（高位对齐模式）

类同标准 IIS-BUS 格式，可通过下图（图8-2）比对它们之间的差异。

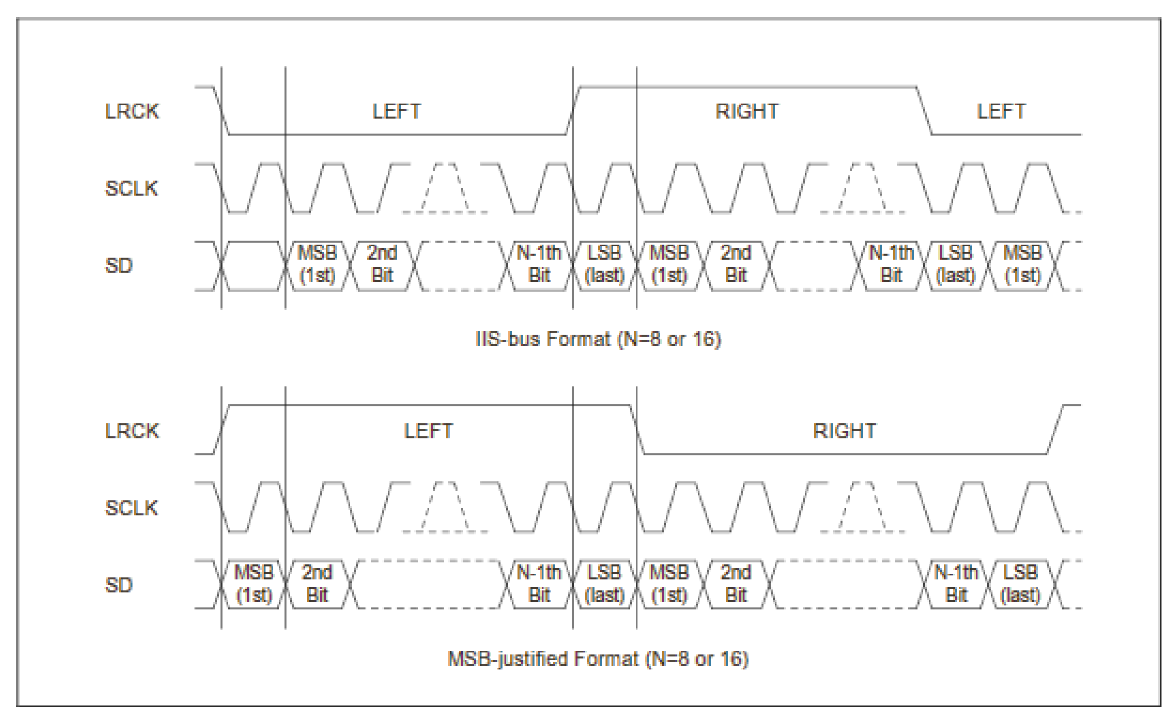


图8-2

采样频率和**DAC**主时钟之间的关系

IIS 的采样频率其实就是 IISLRCK 的频率，DAC 的工作频率必须是 IISLRCK的 256 倍，或 384 倍。如图8-3。

IISLRCK (fs)	8.000 kHz	11.025 kHz	16.000 kHz	22.050 kHz	32.000 kHz	44.100 kHz	48.000 kHz	64.000 kHz	88.200 kHz	96.000 kHz
CODECLK (MHz)	256fs									
	2.0480	2.8224	4.0960	5.6448	8.1920	11.2896	12.2880	16.3840	22.5792	24.5760
	384fs									
	3.0720	4.2336	6.1440	8.4672	12.2880	16.9344	18.4320	24.5760	33.8688	36.8640

图8-3

注：语音资料通常8K采样率即可满足要求；如果CD级音质的采样率为44.1K

三、S3C2410 IIS 控制器寄存器详解

IISCON **：IIS 总线配置寄存器。**见图8-4

IIS Interface	：	IIS 总线使能控制位元
IIS prescaler	：	IIS 主时钟预分频器使能位
Receive channel idle command	：	暂停接收数据控制位
Transmit channel idle command	：	暂停发送数据控制位
Receive DMA service request	：	DMA 方式接收资料使能
Transmit DMA service request	：	DMA 方式发送资料使能
Receive FIFO ready flag	：	接收 FIFO 就绪状态位
Transmit FIFO ready flag	：	发送 FIFO 就绪状态位
Left/Right channel index	：	指示当前发送的是左声道资料，还是右声道资料

IISCON	Bit	Description	Initial State
Left/Right channel index (Read only)	[8]	0 = Left 1 = Right	1
Transmit FIFO ready flag (Read only)	[7]	0 = empty 1 = not empty	0
Receive FIFO ready flag (Read only)	[6]	0 = full 1 = not full	0
Transmit DMA service request	[5]	0 = Disable 1 = Enable	0
Receive DMA service request	[4]	0 = Disable 1 = Enable	0
Transmit channel idle command	[3]	In Idle state the IISLRCK is inactive (Pause Tx). 0 = Not idle 1 = Idle	0
Receive channel idle command	[2]	In Idle state the IISLRCK is inactive (Pause Rx). 0 = Not idle 1 = Idle	0
IIS prescaler	[1]	0 = Disable 1 = Enable	0
IIS interface	[0]	0 = Disable (stop) 1 = Enable (start)	0

图8-4

IISMOD：**IIS 模式控制器**（图8-5）

Serial bit clock freq select ：串行位时钟选择
Master clock freq select ：主时钟频率选择
Serial data bit per channel ：每声道的资料位选择
Serial interface format ：选择 IIS 格式
Active level of left/right ：定义左声道对应的 IISLRCK 的电平
Transmit/reveive mode select：IIS 发送/接收模式选择

00：不进行任何发送
01：接收模式
10：发送模式
11：接收、发送双工模式

Master/Slave mode select ：主/从模式选择

IISMOD	Bit	Description	Initial State
Master/slave mode select	[8]	0 = Master mode (IISLRCK and IISCLK are output mode). 1 = Slave mode (IISLRCK and IISCLK are input mode).	0
Transmit/receive mode select	[7:6]	00 = No transfer 01 = Receive mode 10 = Transmit mode 11 = Transmit and receive mode	00
Active level of left/right channel	[5]	0 = Low for left channel (High for right channel) 1 = High for left channel (Low for right channel)	0
Serial interface format	[4]	0 = IIS compatible format 1 = MSB (Left)-justified format	0
Serial data bit per channel	[3]	0 = 8-bit 1 = 16-bit	0
Master clock frequency select	[2]	0 = 256fs 1 = 384fs (fs: sampling frequency)	0
Serial bit clock frequency select	[1:0]	00 = 16fs 01 = 32fs 10 = 48fs 11 = N/A	00

图8-5

IISPSR：**IIS** 总线时钟预分频系数（图8-6）

IISPSR	Bit	Description	Initial State
Prescaler control A	[9:5]	Data value: 0 ~ 31 Note: Prescaler A makes the master clock that is used the internal block and division factor is N+1.	00000
Prescaler control B	[4:0]	Data value: 0 ~ 31 Note: Prescaler B makes the master clock that is used the external block and division factor is N+1.	00000

图8-6

IISFCON：**IIS** 总线 **FIFO** 控制寄存器（图8-7）

Receive FIFO data count　: FIFO 中当前保存的资料量

Transmit FIFO data count　: FIFO 中当前保存的资料量

Receive FIFO　:是否使能接收 FIFO

Transmit FIFO　:是否使能发送 FIFO

Receive FIFO access mode　:选择 FIFO 的操作模式（普通模式还是DMA）

Transmit FIFO access mode　:选择 FIFO 的操作模式（普通模式还是DMA）

IISFCON	Bit	Description	Initial State
Transmit FIFO access mode select	[15]	0 = Normal 1 = DMA	0
Receive FIFO access mode select	[14]	0 = Normal 1 = DMA	0
Transmit FIFO	[13]	0 = Disable　1 = Enable	0
Receive FIFO	[12]	0 = Disable　1 = Enable	0
Transmit FIFO data count (Read only)	[11:6]	Data count value = 0 ~ 32	000000
Receive FIFO data count (Read only)	[5:0]	Data count value = 0 ~ 32	000000

图8-7

IISFIF：**IIS** 总线资料寄存器（见图8-8）

IISFIF	Bit	Description	Initial State
FENTRY	[15:0]	Transmit/Receive data for IIS	0x0

图8-8

四、UDA1341TS：IIS 音频 DAC 器件简介

UDA1341TS 是 Philips 公司的 IIS 音频 DAC，可以将接收到的资料转换成模拟音频量，或将外部的模拟音频信号转换为响应的资料。它的特性如下：

（1）　低功耗

（2）　3.3V 供电，与 S3C2410 接口方便，不存在电平转换的问题

（3）　体积细小

（4）　内置包括AGC（自动增益控制）模拟前端处理功能

UDA1341TS 的内部逻辑框图如下（图8-9）：

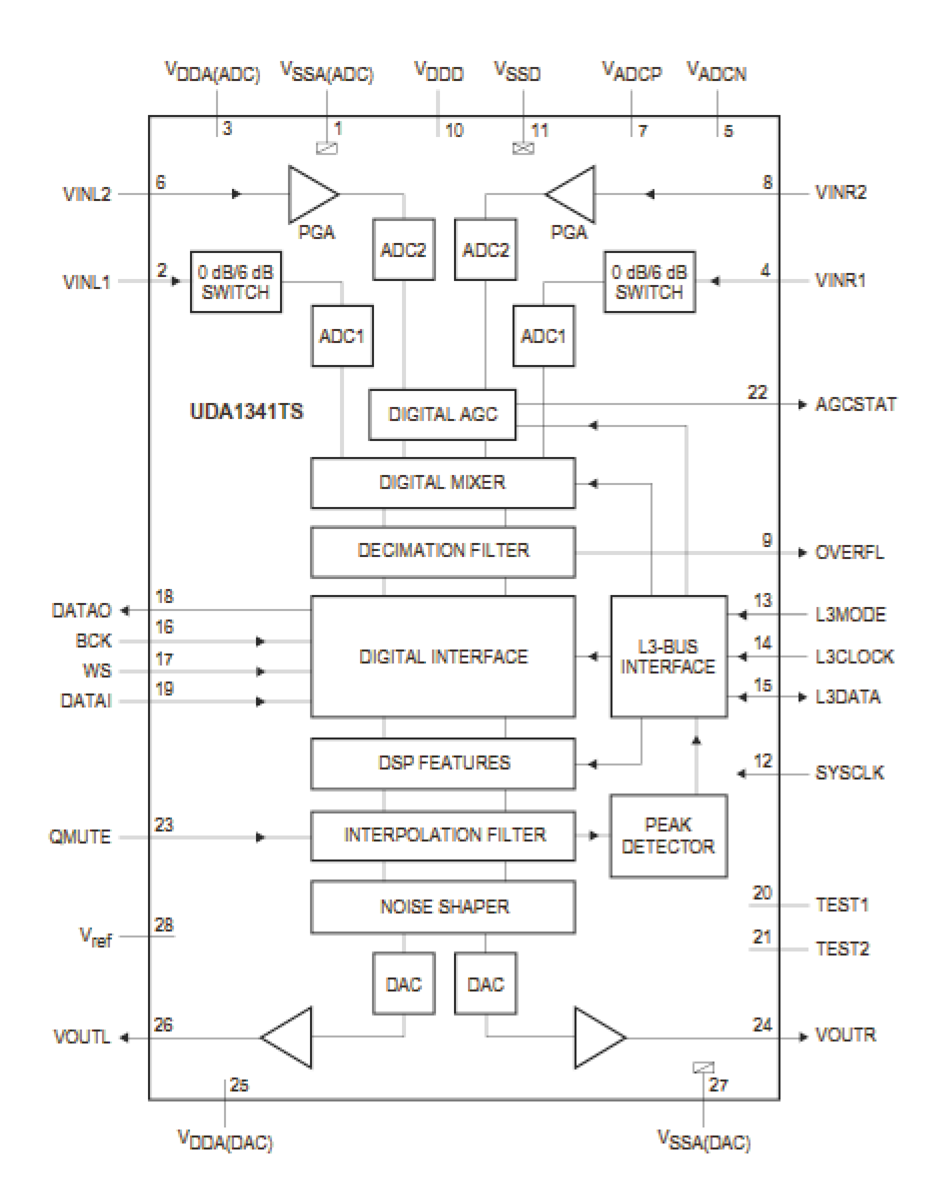


图8-9

同时 UDA1341TS 支持多种数字音频格式，如图8-10。关于 UDA1341TS 的更详细的内容已经超出本书的范围。若有兴趣，可登陆 Philips 的官方网站，查阅有关资料。

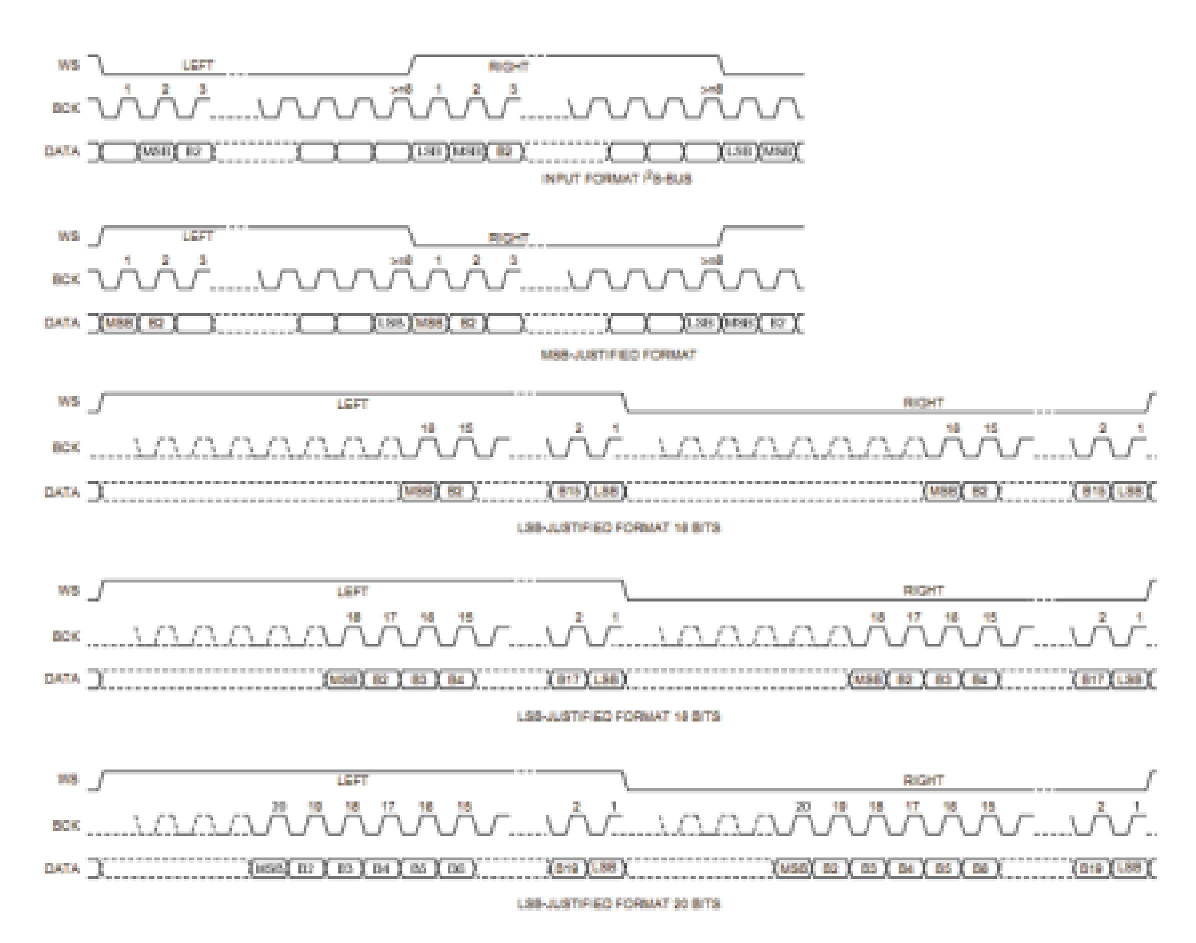

WS
BCK
DATA
LEFT
RIGHT
INPUT FORMAT I²S-BUS
MSB-JUSTIFIED FORMAT
LSB-JUSTIFIED FORMAT 16 BITS
LSB-JUSTIFIED FORMAT 18 BITS
LSB-JUSTIFIED FORMAT 20 BITS

图8-10

第九章 USB1.1 协议及设备

一、USB1.1 概述

USB 是 Universal Serial Bus 的简称。它是一种可以同时处理计算机与具有 USB 接口的多种外设之间通信的电缆总线。这些连接到计算机上的外设共同分享 USB 的带宽。USB 的分时处理机制真正在硬件的意义上实现了计算机外设的即插即用。

如果留心一下当前市场上的计算机外设，大家会发现采用 USB 设备的产品正在逐渐增加。键盘、鼠标、MODEM、游戏杆、音箱、扫描仪等，以前插在串行、并行等外部扩展接口上的部件，甚至一些以前要连接到计算机内部扩展槽上的设备，都开始以 USB 接口的接口出现，USB 设备的发展势头正如日中天。

本章将从技术的角度来探讨一下 USB，有关它的部分请参阅 USB1.1 SPEC （www.usb.org） 的相关文章。

一个基于计算机的 USB 系统可以在系统层次上被分为三个部分：即 USB 主机（USB Host）、USB 器件（USB Device）和 USB 的连接。

所谓 USB 连接实际上是指一种 USB 器件和 USB 主机进行通信的方法。它包括：

（1） 总线的拓扑（由一点分出多点的网络形式）：即外设和主机连接的模式。

（2） 各层之间的关系：即组成 USB 系统的各个部分在完成一个特定的 USB 任务时，各自之间的分工与合作。

（3） 数据流动的模式：即 USB 总线的数据传输方式。

（4） USB 的“分时复用”：因为 USB 提供的是一种共享连接方式，因而为了进行资料的同步传输，致使 USB 对资料的传输和处理必须采用分时处理的机。

USB 的总线拓扑如图9-1所示，在 USB 的树形拓扑中，USB 集线器（HUB）处于节点（Node）的中心位置。而每一个功能部件都和 USB 主机形成唯一的点对点连接，USB 的 HUB 为 USB 的功能部件连接到主机提供了扩展的接口。利用这种树形拓扑，USB 总线支持最多 127 个 USB 外设同时连接到主计算机系统。

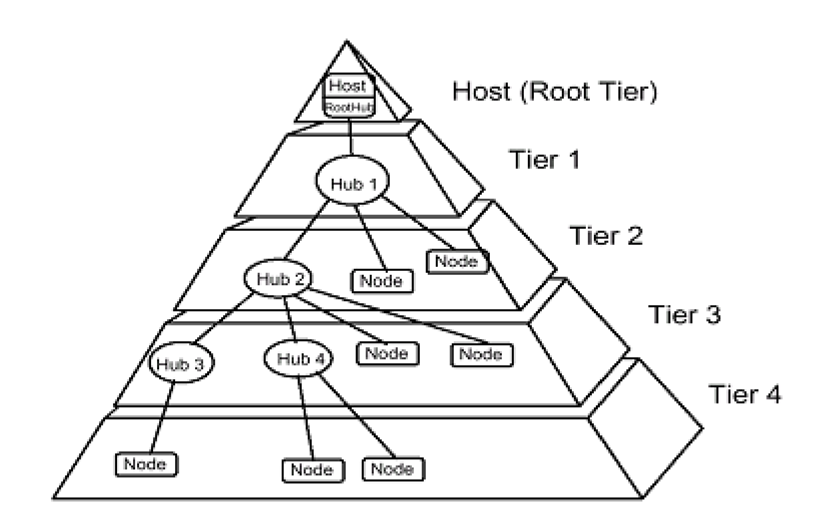


图 9-1 USB 的总线拓扑示意图

一个 USB 系统仅可以有一个主机，而为 USB 器件连接主机系统提供主机接口的部件被称为 USB 主机控制器。 USB 主机控制器是一个由硬件、软件和固件（Firmware）组成的复合体。一块具有 USB 接口的主板通常集成了一个称为 ROOT HUB (根集线器)的部件，它为

主机提供一到多个可以连接其它 USB 外设的 USB 扩展接口,我们通常在主板上见到的 USB 接口都是由 ROOT HUB 提供的。

USB 器件可以分为两种:即 USB HUB 和 USB 功能器件(Function Device)。作为 USB 总线的扩展部件,USB HUB(图 9-2)必须满足以下特征:

1) 为自己和其它外设的连接提供可扩展的下行和上行(Downstream and Upstream)埠。
2) 支持 USB 总线的电源管理机制。
3) 支持总线传输失败的检测和恢复。
4) 可以自动检测下行埠外设的连接和摘除,并向主机报告。
5) 支持低速外设和高速外设的同时连接。

从以上要求出发,USB HUB 在硬件上由两部分组成:HUB 应答器(HUB Repeater)和 HUB 控制器(HUB Controller)。 HUB 应答器响应主机对 USB 外设的设置,以及对连接到它下行端口的 USB 功能部件的连接和摘除(Attached and Detached)的检测、分类,并将其端口信息传送给主机,它也负责如“总线传输失败检测”这样的错误处理;而 HUB 控制器则提供主机到 HUB 之间数据传输的物理机制。如同我们所熟知的大多数计算机外设一样,USB HUB 也有一个用来向主机表明自己身份的“BIOS”系统。这块位于 USB HUB 上的 ROM,通过 USB 特征字使主机可以配置这个 USB HUB,并监控它的每一个埠。

USB 功能器件即可以为主机系统提供某种功能的 USB 器件,如一个 USB ISDN 的调制解调器、或是一只 USB 接口的数字摄像机、 USB 的键盘或鼠标等。

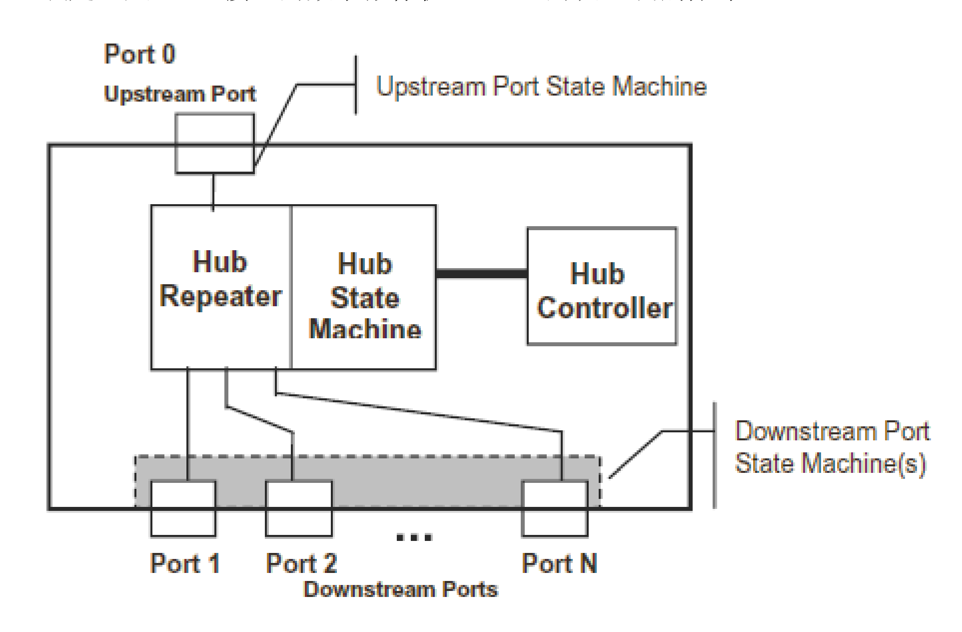


图 9-2 USB HUB 的结构

图 9-3 一个典型的 USB 功能器件结构框图

USB 的功能器件作为 USB 外设(USB Function)，它必须保持和 USB 协议的完全兼容，并可以响应标准的 USB 操作。同样，用于表明自己身份的“BIOS”系统对于 USB 外设也是必不可少的，这在 USB 外设上被称为协议层。在物理机制上，一个 USB 外设可以由四部分构成(见图 9-3):

1) 用于实现和USB协议兼容的SIE部分。
2) 用于内存件特征字、存储实现外设特殊功能程序及厂家信息的协议层(ROM)。
3) 用于实现外设功能的传感器及对资料进行简单处理的 DSP 部分。
4) 将外设连接到主机或 USB HUB 的接口部分。

根据传输率的不同， USB 器件被分为高速和低速两种。低速外设的标准传输率为1.5Mbps，而高速外设的标准传输率为12Mbps。所有的 USB HUB 都为高速外设，而功能部件则可以根据外设的具体情况设计成不同的传输率，如用于视频、音频传输的外设大都采用12Mbps 的传输率，而像键盘、鼠标这样的人机输入设备(HID)则设计成低速外设。由于 USB 的数据传输采用资料包的形式，因而使得连接到主机的所有的 USB 外设可以同时工作而互不干扰。不幸的是，所有这些 USB 外设必须同时分享 USB 协议所规定的 USB 带宽(这个带宽在USB 1.0协议中为12Mbps)，虽然 USB 的分时处理机制可以使有限的 USB 带宽在各设备之间动态地分配，但如果两台以上的高速外设同时使用这样的连接方法，就会使它们都无法享用到最高的 USB 带宽，从而降低了性能。这也正是 Intel 这样的巨头为什么要推出USB 2.0 协议的原因(在 USB 2.0协议中 USB 的总线带宽一下子被提高到了 480Mbps)。

用于实现外设到主机或 USB HUB 连接的是USB线缆(图 9-4)。从严格意义上讲， USB 线缆应属于 USB 器件的接口部分。USB 线缆由四根线组成，其中一根是电源线 VBus，一根是地线 GND，其余两根是用于差动信号传输的资料线(D+，D-)。将数据流驱动成为差动信号来传输的方法可以有效提高信号的抗干扰能力(EMI)。在资料线末端设置结束电阻的思路是非常巧妙的，以至对于 HUB 来判别所连接的外设是高速外设或是低速外设，仅仅只需要检测在外设被初次连接时， D+ 或 D- 上的信号是高或是低即可。因为对于 USB 协议来讲，要求低速外设在其 D-端并联一个1.5kΩ的接地电阻，而高速外设则在 D+ 端接同样的电阻。在加电时，根据低速外设的 D- 线和高速外设的 D+ 线所处的状态， HUB 就很容易判别器件的种类，从而为器件配置不同的信息。图 9-5 表明了一个典型的高速外设的连接状况。为提高数据传输的可靠性、系统的兼容性及标准化程度， USB 协议对用于 USB 的线缆提出了较为严格的要求。如用于高速传输的 USB 线缆，其最大长度不应超过 5 米，而用于低速传输的线缆则最大长度为 2 米，每根资料线的电阻应为标准的 90Ω。

USB 系统可以通过 USB 线缆为其外设提供不高于 +5V 、500mA 的总线电源。那些完全依靠 USB 线缆来提供电源的器件被称为总线供电器件(Bus-powered device),而自带电源

的器件则被称为自供电外设(Self-powered device)。需要注意的是，当一个外设初次连接时，器件的配置和分类并不使用外设自带的电源，而是通过 USB 线缆提供的电源来使外设处于上电状态。

无论在软件还是硬件层次上， USB 主机都处于 USB 系统的核心。主机系统（图 9-6）不仅包含了用于和 USB 外设进行通信的 USB 主机控制器及用于连接的 USB 接口（SIE），更重要的是主机系统是 USB 系统软件和 USB 客户软件的载体。

总而言之，USB 主机软件系统可以分为三个部分:

1) 客户软件部分(CSW)，在逻辑上和外设的功能部件部分进行资料的交换
2) USB 系统软件部分（即 HCDI），在逻辑和实际中作为 HCD 和 USBD 之间的接口
3) USB 主机控制器软件部分（即 HCD 和 USBD），用于对外设和主机的所有 USB 有关部分的控制和管理，包括外设的 SIE 部分、 USB 资料发送接收器(Transreceiver)部分及外设的协议层等。

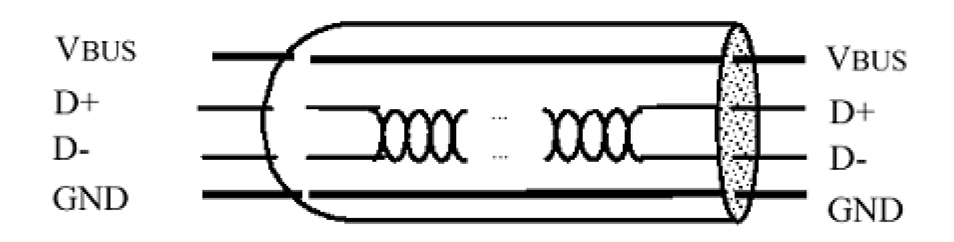


图 9-4 USB 线缆

图 9-5 高速外设的 USB 线缆和电阻的连接图

图 9-6 USB 主机系统的结构及各部分之间的关系

当一个 USB 外设初次接入一个 USB 系统时，主机就会为该 USB 外设分配一个唯一的 USB 地址，并作为该 USB 外设的唯一标识(USB 系统最多可以分配这样的地址 127 个)，这称为 USB 的总线枚举(Bus Enumeration)过程。USB 使用总线枚举方法在计算机系统运行期间动态检测外设的连接和摘除，并动态地分配 USB 地址，从而在硬件意义上真正实现“即插即用”和“热插拔”。

在所有的 USB 信道之间动态地分配带宽是 USB 总线的特征之一。当一台 USB 外设在连接(Attached)并配置(Configuration)以后，主机即会为该 USB 外设的信道分配 USB 带宽；而当该 USB 外设从 USB 系统中摘除(Detached)或是处于挂起(Suspended)状态时，则它所占用的 USB 带宽即会被释放，并为其它的 USB 外设所分享。这种“分时复用”(Scheduling the USB)的带宽分配机制大大地提高了 USB 带宽利用率。

作为一种先进的总线方式，USB 提供了基于主机的电源管理系统。USB 系统会在一台外设长时间(这个时间一般在 3.0ms 以上)处于非使用状态时自动将该设备挂起(Suspend)，当一台 USB 外设处于挂起状态时， USB 总线通过 USB 线缆为该设备仅仅提供 500μA 以下的电流，并把该外设所占用的 USB 带宽分配给其它的 USB 外设。 USB 的电源管理机制使它支持如远程唤醒这样的高级特性。当一台外设处于挂起状态(Suspended Mode)时，必须先通过主机使该设备“唤醒”(Resume)，然后才可以执行 USB 操作。

USB 的这种智能电源管理机制，使得它特别适合如笔记本计算机之类的设备的应用。

我们知道，USB 总线是一种串行总线，即它的资料是一个 bit 一个 bit 来传送的。虽然 USB 总线是把这些 bit 形式的资料打成资料包来传送，但资料的同步也是必不可少的。USB 1.0/1.1 协议规定， USB 的标准脉冲时钟为 12MHz ，而其总线时钟为 1ms ，即每隔 1ms ，USB 器件应为 USB 线缆产生一个时钟脉冲序列。这个脉冲序列称为帧开始资料包(SOF，如图 9-7 所示)，主机利用 SOF 来同步 USB 资料的发送和接收。

图 9-7 帧开始资料包在 USB 数据传输中的分布

由此可见,对于一个数据传输率为 12Mbps 的外设而言,它每一帧的长度为 12000bit ；而对于低速外设而言，它每一帧的长度仅有 1500bit。 USB 总线并不关心外设的数据采集系统及其处理的速率，无论对于怎样的资料产生或是接收，它总是以外设所事先规定的 USB 标准传输率来传输资料。这就要求外设厂商必须在数据采集或接收系统和 USB 协议系统(SIE)之间，设置大小合适的先入先出模式(FIFO)来对资料进行缓存。

在 USB 系统中，资料是通过 USB 线缆采用 USB 资料包从主机传送到外设或是从外设传送到主机的。在 USB 协议中，把基于外设的资料源和基于主机的资料接收软件(或者方向相反)之间的数据传输模式称为信道或管道(Pipe)。信道分为流模式的信道(Stream Pipe)和消息模式的信道(Message Pipe)两种。信道和外设所定义的资料带宽、数据传输模式以及外设的功能部件的特性(如缓存大小、数据传输的方向等)相关。只要一个 USB 外设一经连接，就会在主机和外设之间建立信道。对于任何的 USB 外设，在它连接到一个 USB 系统中，并被 USB 主机经 USB 线缆加电使其处于上电状态时，都会在 USB 主机和外设的协议层之间首先建立一个称为 Endpoint 0（端点 0）的消息信道，这个信道又称为控制信道，主要用于外设的配置(Configuration)、对外设所处状态的检测及控制命令的传送等。信道方式的结构使得 USB 系统支持一个外设拥有多个功能部件(用 Endpoint 0、Endpoint 1...Endpoint n 这样的方法进行标识)，这些功能部件可以同时地、以不同的数据传输方向在同一条 USB 线缆上进行数据传输而互不影响（图 9-8）。比如一个 USB 的 ISDN MODEM ，就可以同时拥有一个上传的信道和一个下载的信道，并能同时很好地工作。

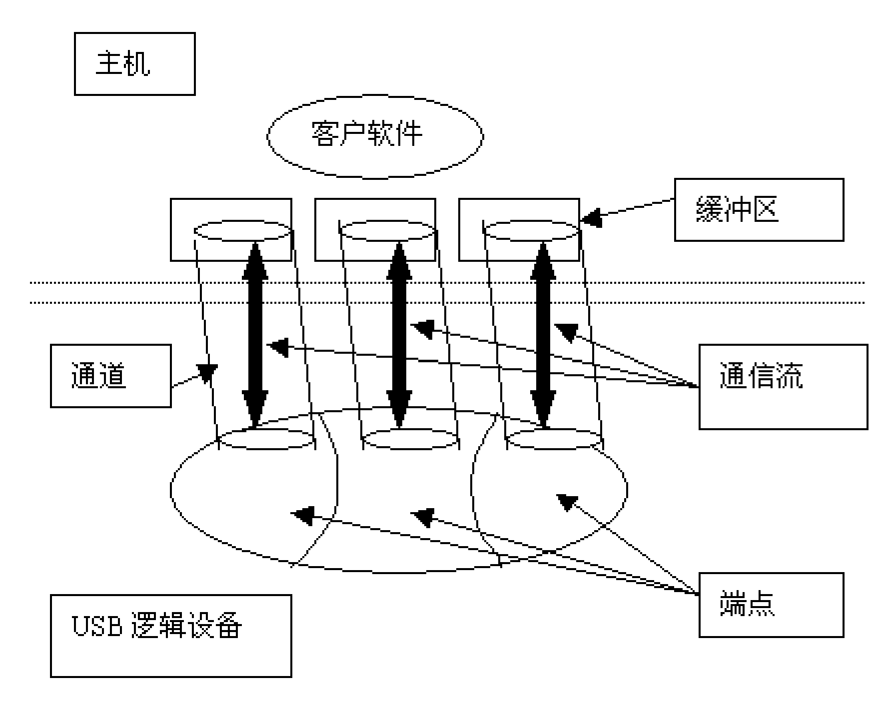


图 9-8 USB 的通信流及信道

为实现多外设、多信道地同时工作,USB 总线使用资料包的方式来传输资料和控制信息。USB 数据传输中的每一个资料包都以一个同步字段开始（图 9-9），它的最后两个 bit 作为 PID 字段开始的标志。紧跟在同步字段之后的一段 8bit 的脉冲序列称为 PID （资料包标识字段，如图 9-10 所示）， PID 字段的前四位用来标记该资料包的类型，后四位则作为对前四位的校验。 PID 字段被分为标记 PID （共有 IN 、 OUT 、 SETUP 或 SOF 四种)、资料 PID （ DATA0 或 DATA1 ）、握手 PID（ ACK、 NAK 或 STALL ）及特殊 PID 等。主机根据 PID 字段的类型来判断一个资料包中所包含的数据类型，并执行相应的操作。

图 9-9 同步字段

图 9-10 PID 字段

当一个 USB 外设初次连接时， USB 系统会为这台外设分配唯一的 USB 地址，这个地址通过地址寄存器（ ADDR)来标记,以保证资料包不会传送到别的 USB 外设。7bit 的 ADDR 使得 USB 系统最大寻址为 127 台设备（ ADDR 字段，如图 9-11 所示）。由于一台 USB 外设可能具有多个信道，因而在 ADDR 字段后会有一个附加的端点字段（ Endpoint Field，简标为 ENDP ）来标记不同的信道（图 9-12）。所有的 USB 外设都必须支持 Endpoint 0 信道，用 0000 来标记。对于高速设备，可以最大支持 16 个信道，而低速设备在 Endpoint 0 之外仅能有一个信道。

图 9-11 资料包的 ADDR 字段

图 9-12 端点(Endpoint)字段

数据域位作为一次 USB 数据传输的中心目的，在一个 USB 资料包中可以包含 0～1203 Byte 的资料(图 9-13)。而帧数量字段则包含在帧开始资料包中，对有的应用场合，可以用帧数量字段作为资料的同步信号。

为保证控制、块传送及中断传送中资料包的正确性， CRC 校验字段被引用到如标记、资料、帧开始（ SOF ）这样的资料包中。 CRC 校验（资料冗余校验）可以给予资料以 100% 的正确性检验。

图 9-13 USB 的数据域位

在 USB 系统中，有四种形式的资料包：信令包（ Token Packets）、DATA 资料包（DATA Packets）、帧开始包(SOF Packets)和握手包(Handshake Packets)。

（1） 信令包由 PID 、ADDR 、ENDP 和 CRC5 四个字段组成(图 9-14)。它因为 PID 字段的不同而分为输入类型(IN)、输出类型(OUT)和设置类型(SETUP)三种。信令包处于每一次 USB 传输的 DATA 资料包前面，以指明这次 USB 操作的类型(PID 字段标记)、操作的对象(在 ADDR 和 ENDP 字段中指明)等信息。 5bit 的 CRC 校验位用来确保标记资料包的正确性。

图 9-14 标记数据包的组成

（2） 我们已经指出， USB 主机会每隔 1ms 在 USB 总线上产生一个 SOF 的 USB 帧同步信号， SOF 资料包包含了这个脉冲序列的实际内容(图 9-15)，它由 SOF 格式的 PID 字段、帧数量字段和 5bit 的 CRC 校验码组成。主机利用 SOF 资料包来同步资料的传送和接收。

图 9-15 SOF 资料包的格式

（3） 用于传输真正资料的 DATA 资料包(图 9-16)，因为 PID 的不同可以分为 DATA0 和 DATA1 两种。 DATA0 为偶数据包，DATA1 为奇数据包。DATA 资料包的奇偶性分类易于资料的双流水处理，而用于控制传输的 DATA 资料包总是以 DATA0 来传送资料。

图 9-16 DATA 资料包的格式

（4） 握手资料包仅仅包含一个 PID 字段(图 9-17)， ACK 形式的 PID 表明此次 USB 传输没有发生错误，资料已经成功的传输；而 NAK 形式的握手资料包则向主机表明此次 USB 传输因为 CRC 校验错误或别的原因而失败了，从而使得主机可以进行资料的重新传输； STALL 形式的响应向主机报告外设此刻正处于挂起状态而无法完成资料的传输。

图 9-17 握手资料包

需要指出的是,每个资料包的结束都会有两个 bit 宽的 EOP 字段作为资料包结束的标志(图 7-18)，EOP 在差模信号中表现为 D+ 和 D- 都处于“0”状态。对于高速 USB 外设而言，这个脉冲宽度在 160～175ns 之间，而低速设备则在 1.25～1.50μs 之间。无论其后是否有其它的资料包，USB 线缆都会在 EOP 字段后紧跟 1bit 的总线空闲位。USB 主机或外设利用 EOP 来判断一个资料包的结束。

图 9-18 EOP 字段在差模信号中的电压表现

在前面我们已经提到，每一个 USB 信道对应着一个特定的 USB 传输模式，根据不同的需要，USB 外设可以为 USB 信道指定不同的 USB 传输模式。USB 总线支持四种数据传输模式：

（1） 控制传输模式，控制传输用于在外设初次连接时对器件进行配置；对外设的状态进行实时检测；对控制命令的传送等；也可以在器件配置完成后被客户软件用于其它目的。Endpoint 0 信道只可以采用控制传送的方式。

（2） 块传送模式(图 9-19)，块传送用于进行批量的、非实时的数据传输。如一台 USB 扫描仪即可采用块传送的模式，以保证资料连续地、在硬件层次上的实时纠错地传送。采用块传送方式的信道所占用的 USB 带宽，在实时带宽分配中具有最高的优先级。

图 9-19 块传送的流程

（3） 同步传输模式(图 9-20)，同步传输适用于那些要求资料连续地、实时地、以固定的数据传输率产生、传送并消耗的场合，如数字录像

机等。为保证数据传输的实时性，同步传输不进行资料错误的重试，也不在硬件层次上响应一个握手资料包,这样有可能使数据流中存在资料错误的隐患。为保证在同步传输数据流中致命错误的几率小到可以容忍的程度,而数据传输的延迟又不会对外设的性能造成太大的影响，厂商必须为使用同步传输的信道选择一个合适的带宽(即必须在速度和品质之间做出权衡)。

图 9-20 同步传输的流程

（4） 中断传输模式(图 9-21)，对于那些小批量的、点式、非连续的数据传输应用的场合，如用于人机交互的鼠标、键盘、游戏杆等，中断传输的方式是最适合的。

图 9-21 中断传输的流程

上述内容并不想详细论述 USB 外设（本部分所说的USB外设如无特别说明均指USB功能器件）的设计细节，而只想介绍 USB 功能器件的一般组成，以此来帮助读者了解 USB 外设的基本软硬件构成，以便了解 USB 外设的工作过程和原理。

组成外设的传感器件和 DSP 因为外设的具体应用各异而有所不同。如对于一台 CMOS 数字摄像头，它的 CMOS 光电耦合器及其 DSP 部分并不因为使用什么样的接口方式而有所改变(如早期的摄像头皆采用 ECP 的并口增强模式来进行图像数据的传输,而现在几乎都是 USB 接口)。因而本文的重点是阐述 USB 外设接口的部分，即 USB Device Microcontroller (USB 器件微控制器)。

USB 总线是以差模驱动的方式来进行数据传输的，但在资料包发送之前， USB 协议规定必须使用 NRZI 的编码方式来对资料进行编码。当然，在 USB 外设中，用于译码的器件对外设来说也是必不可少的。 NRZI 的编码协议其实很简单，它采用的是逢“1”保持，逢“0”跳变的原则(图 9-22)，而 NRZI 的译码则采用相反的操作。

图 9-22 NRZI 数据编码

为保证数据流中有足够的信号变化， USB 协议规定了 Bit stuffing (加填充位)的原则，即如果信号流中连续出现六位以上的资料“1”，则每隔六位，必须插入一个“0”，然后才进入 NRZI 编码。图 9-23 是一串原始资料及其加填充位后和 NRZI 编码后的资料格式对比。

图 9-23 原始资料和加填充位后及 NRZI 编码后的资料格式对比

SIE (Serial interface Engine)是 USB 外设最重要的硬件组成部分之一，它主要由四部分组成：

1) 硬件上用来完成 NRZI 编/译码和加/去填充位操作的，NRZI/Bit Buffing 和 NRZO/Bit Unstuffing 的部分。
2) 硬件上产生资料的 CRC 校验码并对资料包进行 CRC 校验的 CRC check & Generator 部分。
3) 用来将并行资料转化成 USB 串行资料的并/串转换部分(Packet Encode)，将主机发送的 USB 资料包转化成可以识别的并行资料的串/并转换部分(Packet Decode)。
4) 检测和产生 SOP (即每个资料包的同步字段)和 EOP 信号的部分。

USB 外设使用一段代码来存储关于该外设工作的一些重要信息，这被称为 USB 的协议层(Protocol Layer)，它不仅存储了诸如厂家识别号、该外设所属的类型(是 HUB 还是 Function，是低速还是高速设备)、电源管理等常规信息，更重要的是还存储了外设的设备类型、器件配置信息、功能部件的描述、接口信息等,其存储方式都采用特征字(Descriptors)的方式。 USB 主机通过在外设的协议层和主机之间建立 Endpoint 0 信道、采用控制传输的方式对这些信息进行存取。特征字采用 USB 协议所规定的结构和代码排列(关于特征字的详细信息请参阅 USB 协议标准)。协议层是一台 USB 外设能够被主机正确识别和配置，并正常工作的前提。可以说，协议层是一台 USB 外设的固件(Firmware)中心。

我们知道，资料采样率因采样精度和使用的不同场合而不同，如对于音频应用，就可以采用 22.05kHz 或 44.1kHz 的采样率，而这个时钟并不和 USB 标准时钟对应。因而在实际应用中，为保证采集到的资料无丢失地打包和传送，必须在 SIE 和数据采集部件(对诸如音箱或打印机等外设则为资料消耗部件)之间设立 FIFOs，以便对资料进行缓存。对于采用块传送和同步传送的外设而言， FIFOs 的作用显得尤为重要。例如一台采用同步传输的 USB 数字摄像机(现在市场上有很多这种类型的产品)，我们假设它的 CCD 为 400×300 像素，

那么为保证资料正确地压缩、传输和接收，直到以后的解压缩及处理，在动态采集中，FIFOs 至少要存储一帧图像，即要求 FIFOs 有 400 × 300 = 12KB 的容量。

在 USB 外设中,用于实现和 USB 线缆无缝连接的 USB 传输接收部分(Transreceiver)是必不可少的，它不仅要在电气和物理层面上实现和 USB 线缆的连接，而且要完成对资料包的差模驱动或分离的操作。

以上我们简述了 USB 外设接口的硬件组成，那么在完成 USB 数据传输的过程中，这些硬件又是如何配合工作并和位于主机的软硬件交互，以完成数据传输的呢?

前面已经提到， USB 总线采用总线列举的方法来标记和管理外设所处的状态，当一台 USB 外设初次连接到 USB 系统中后，通过 8 个步骤来完成它的初始化：

1) USB 外设所连接的 HUB (ROOT HUB 或扩展 HUB)检测到所连接的 USB 外设并自动通知主机,以及它的端口状态的变化,这时外设还处于禁止(Disabled)状态。
2) 主机通过对 HUB 的查询以确认外设的连接。
3) 现在，主机已经知道有一台新的 USB 外设连接到了 USB 系统中，然后，它激活(Enabled)这个 HUB 的埠，并向 HUB 发送一个复位(Reset)该埠的命令。
4) HUB 将复位信号保持 10ms ，为连接到该埠的外设提供 100mA 的总线电流，这时该外设处于上电状态，它的所有寄存器被清空并指向默认的地址。
5) 在外设分配到唯一的 USB 地址以前，它的默认信道均使用主机的默认地址。然后主机通过读取外设协议层的特征字来了解该外设的默认信道所使用的实际的最大资料有效载荷宽度（即外设在特征字中所定义的在 DATA0 资料包中数据域位的长度）。
6) 主机分配一个唯一的 USB 地址给该外设，并使它处于 Addressed 状态。
7) 主机开始使用 Endpoint 0 信道读取外设的器件配置特征字，这会花去几帧的时间。
8) 基于器件配置特征字，主机为该外设指定一个配置值，这时，外设即处于配置(Configured)状态了，它所有的端点(Endpoint)这时也处于配置值所描述的状态。从外设的角度来看，这时该外设已处于准备使用的状态。

在一台外设能被使用之前，它必须被配置。“配置”即主机根据外设的配置特征字来定义器件的配置寄存器，以便规定外设的所有 Endpoint 的工作环境。如某信道所采用的数据传输方式，该外设所属的器件“类”(Class)、“子类”(SubClass)等，从而通过基于主机的 USB 系统软件或客户软件对外设进行控制。当一台 USB 外设配置好以后，即会进入到挂起 (Suspend)状态，直到它开始被使用。

必须指出的是，一台 USB 外设一旦配置好，它的每一个特定的信道只能使用一种数据传输方式。 Endpoint 0 信道只能采用控制传送的方式，主机通过 Endpoint 0 来传送标准的 USB 命令，完成诸如读取器件配置特征字、控制外设对资料的采集、处理和传送等任务，并可以通过 Endpoint 0 来检测和改变外设所处的状态(如对外设的远程唤醒、挂起和恢复等)。

对于一台采用同步传输的数字摄像机来说，数据传输的过程如下：

（1）应用软件(CSW)在内存中开辟资料缓冲区，并通过标准 USB 命令字向外设发出资料请求(IRPs)。
（2）主机 USB 系统软件通过对该 IRPs 的翻译形成 Token 资料包发送到外设，这时主机进入等待状态。
（3）外设对资料包进行 NRZI 译码和 Bit Unstuffing 操作及 CRC 校验，确认后接收主机 PID 字段中所包含的命令并开始采集资料。

（4）采集到的并行资料首先进入 FIFOs ，并通过并/串转换部件形成串行脉冲。

（5）根据器件配置寄存器的要求对资料进行符合条件的分割，配置资料包的 PID 字段等以形成原始资料包。

（6）通过 CRC 校验产生器对每一个资料包生成 CRC 校验码字段，SOP & EOP 信号产生器为该资料包加入同步字段头和资料包结束符。

（7）数据包的 NRZI 编码和 Bit Stuffing 操作。

（8）使用收发器(Transreceiver)将数据流驱动到 USB 线缆上。

（9）主机控制器将 USB 资料转化成为普通资料送到资料缓冲区以进行资料的进一步处理。如果是控制传输、块传输或中断传输方式，在资料被成功传送后，主机还会向外设发送 ACK 的握手资料包作为响应。

图 9-24 简单描绘了异步数据传输的请求和传送过程(在同步传输中没有 Handshake 部分)。

图 9-24 异步数据传输的请求和传送过程

至此，我们已从几个方面较详细地介绍了 USB 系统的软硬件构成及 USB 的数据传输协议。USB 可以说是开辟了计算机外设接口的新纪元。它把人们从繁杂的联机、不同的接口标准和恼人的中断冲突中解放出来；

使“PnP”和“热插拔”这样的特性不再只是口号；它大大扩展了计算机可连接的外设数目；它的智能电源管理有效地降低了手持计算机的电源损耗。USB 正在成为市场的热点，越来越多的外设生产厂家将自己的产品转向 USB 接口。而 USB 2.0 协议的推出，无疑对 USB 技术的发展起到了推波助澜的作用。

在如鼠标、键盘、手写板或是游戏杆等人机交互的应用场合：如扫描仪、数码相机、移动存贮设备、数字摄像机等数据输入应用场合，USB 无疑是替代传统串/并口的最佳接口方式，它们使得 USB 的优点得到了充分发挥。

二、S3C2410 内置 USB1.1 Device 控制器

S3C2410 内置的 USB Device 控制器具有一下特性：

（1） 完全兼容 USB1.1 协议
（2） 支持全速（Full Speed）设备
（3） 集成的 USB 收发器
（4） 支持Control、Interrupt 和 Bulk传输模式
（5） 5 个具备 FIFO 的通讯端点
（6） Bulk 端点支持DMA操作方式
（7） 接收和发送均有 64Byte 的FIFO
（8） 支持挂起和远程唤醒功能

下图（图9-25）是 USB 控制器的内部逻辑示意图

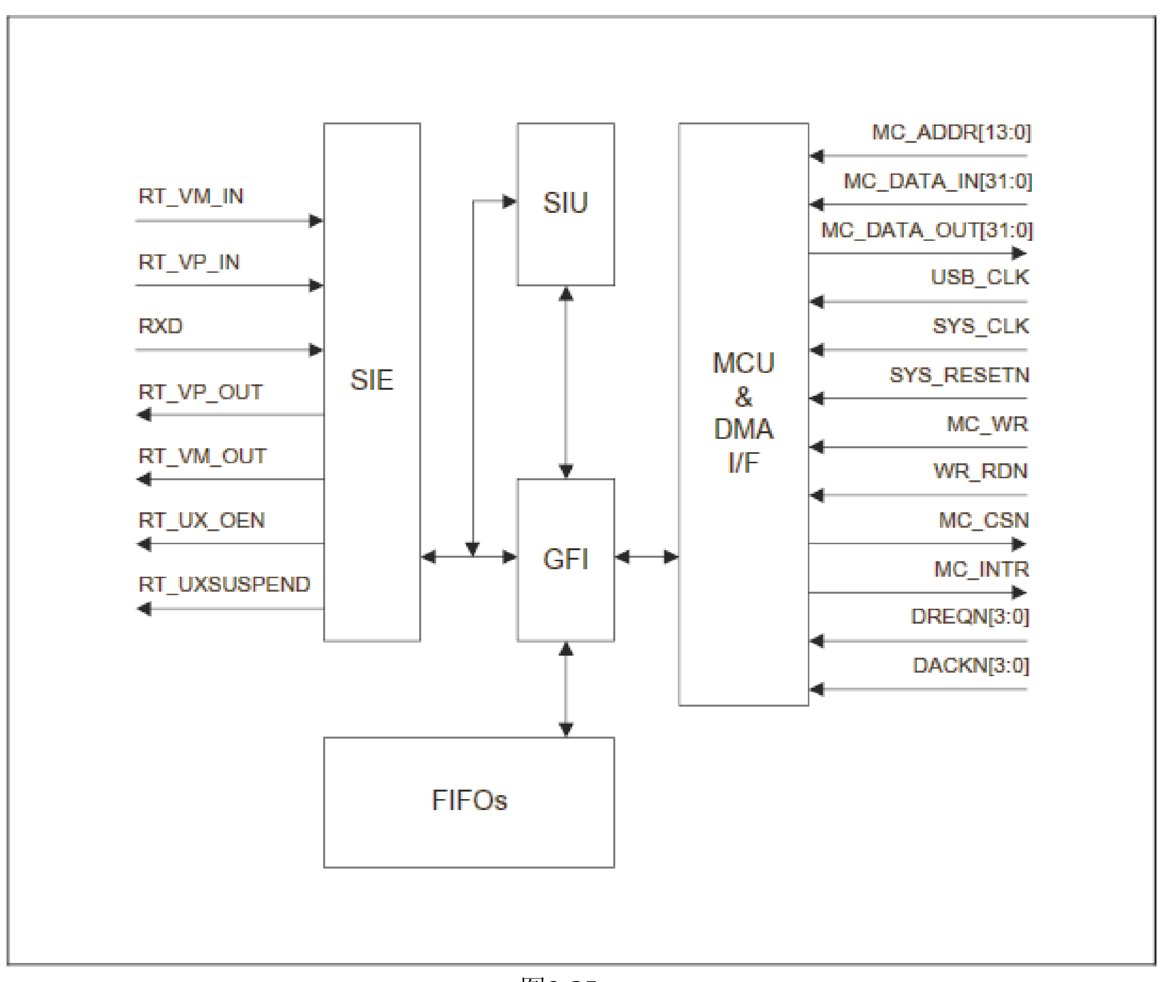


图9-25

三、S3C2410 USB 内部控制寄存器简介：

FUNC_ADDR_REG：**USB 设备地址寄存器**（见图9-26）

FUNCTION_ADDR ：CPU 将由 USB 主机分配的器件地址写入该字段

ADDR_UPDATE ：当 CPU 写入新的地址后，置该位来更新 FUNCTION_ADDR 字段中的值

FUNC_ADDR_REG	Bit	MCU	USB	Description	Initial State
ADDR_UPDATE	[7]	R /SET	R /CLEAR	Set by the MCU whenever it updates the FUNCTION_ADDR field in this register. This bit will be cleared by USB when DATA_END bit in EP0_CSR register.	0
FUNCTION_ADDR	[6:0]	R/W	R	The MCU write the unique address, assigned by host, to this field.	00

图9-26

PWR_ADDR：**电源管理控制寄存器**（见图9-27）

SUSPEND_EN ：使能SUSPEND模式

SUSPEND_MODE：当设备进入 SUSPEND 状态时，由 USB 控制器设置

MCU_RESUME ：由 CPU 设置 进入 RESUME 状态

USB_RESET ：当 USB 主机发出复位（RESET）命令后，由CPU设置

ISO_UPDATE ：只用于 ISO 传输模式

PWR_ADDR	Bit	MCU	USB	Description	Initial State
ISO_UPDATE	[7]	R/W	R	Used for ISO mode only. If set, GFI waits for a SOF token to set IN_PKT_RDY even though a packet to send is already loaded by MCU. If an IN token is received before a SOF token, then a zero length data packet will be sent.	0
Reserved	[6:4]	–	–	–	–
USB_RESET	[3]	R	SET	Set by the USB if reset signaling is received from the host. This bit remains set as long as reset signaling persists on the bus	0
MCU_RESUME	[2]	R/W	R /CLEAR	Set by the MCU for MCU Resume. The USB generates the resume signaling during 10ms, if this bit is set in suspend mode.	
SUSPEND_MODE	[1]	R	SET /CLEAR	Set by USB automatically when the device enter into suspend mode. It is cleared under the following conditions: 1) The MCU clears the MCU_RESUME bit by writing '0', in order to end remote resume signaling. 2) The resume signal form host is received.	0
SUSPEND_EN	[0]	R/W	R	Suspend mode enable control bit 0 = Disable (default). The device will not enter suspend mode. 1 = Enable suspend mode.	0

图9-27

EP_INT_REG： **Endpoint** 中断标志寄存器（见图9-28）

EP_INT_REG	Bit	MCU	USB	Description	Initial State
EP1~EP4 Interrupt	[4:1]	R /CLEAR	SET	**For BULK/INTERRUPT IN endpoints:** Set by the USB under the following conditions: 1. IN_PKT_RDY bit is cleared. 2. FIFO is flushed 3. SENT_STALL set. **For BULK/INTERRUPT OUT endpoints:** Set by the USB under the following conditions: 1. Sets OUT_PKT_RDY bit 2. Sets SENT_STALL bit **For ISO IN endpoints:** Set by the USB under the following conditions: 1. UNDER_RUN bit is set 2. IN_PKT_RDY bit is cleared. 3. FIFO is flushed **NOTE:** Conditions 1 and 2 are mutually exclusive **For ISO OUT endpoints:** Set by the USB under the following conditions: 1. OUT_PKT_RDY bit is set 2. OVER RUN bit is set. **NOTE:** Conditions 1 and 2 are mutually exclusive.	0
EP0 Interrupt	[0]	R /CLEAR	SET	Correspond to endpoint 0 interrupt. Set by the USB under the following conditions: 1. OUT_PKT_RDY bit is set. 2. IN_PKT_RDY bit is cleared. 3. SENT_STALL bit is set 4. SETUP_END bit is set 5. DATA_END bit is cleared (it indicates the end of control transfer).	0

图9-28

USB_INT_REG：**USB中断标志寄存器**（见图9-29）

USB_INT_REG	Bit	MCU	USB	Description	Initial State
RESET Interrupt	[2]	R /CLEAR	SET	Set by the USB when it receives reset signaling.	0
RESUME Interrupt	[1]	R /CLEAR	SET	Set by the USB when it receives resume signaling, *while in Suspend mode*. If the resume occurs due to a USB reset, then the MCU is first interrupted with a RESUME interrupt. Once the clocks resume and the SE0 condition persists for 3ms, USB RESET interrupt will be asserted.	0
SUSPEND Interrupt	[0]	R /CLEAR	SET	Set by the USB when it receives suspend signalizing. This bit is set whenever there is no activity for 3ms on the bus. Thus, if the MCU does not stop the clock after the first suspend interrupt, it will continue to be interrupted every 3ms as long as there is no activity on the USB bus. By default, this interrupt is disabled.	0

图9-29

EP_INT_EN_REG：**Endpoint** 中断使能寄存器（见图9-30）

EP_INT_EN_REG	Bit	MCU	USB	Description	Initial State
EP4_INT_EN	[4]	R/W	R	EP4 Interrupt Enable bit 0 = Interrupt disable 1 = Enable	1
EP3_INT_EN	[3]	R/W	R	EP3 Interrupt Enable bit 0 = Interrupt disable 1 = Enable	1
EP2_INT_EN	[2]	R/W	R	EP2 Interrupt Enable bit 0 = Interrupt disable 1 = Enable	1
EP1_INT_EN	[1]	R/W	R	EP1 Interrupt Enable bit 0 = Interrupt disable 1 = Enable	1
EP0_INT_EN	[0]	R/W	R	EP0 Interrupt Enable bit 0 = Interrupt disable 1 = Enable	1

图9-30

INT_MASK_REG：**USB** 中断屏蔽寄存器（见图9-31）

INT_MASK_REG	Bit	MCU	USB	Description	Initial State
RESET_INT_EN	[2]	R/W	R	Reset interrupt enable bit 0 = Interrupt disable 1 = Enable	1
Reserved	[1]	–	–	–	0
SUSPEND_INT_EN	[0]	R/W	R	Suspend interrupt enable bit 0 = Interrupt disable 1 = Enable	0

图9-31

FRAME_NUM_REG：帧计数器（低位资料）（见图9-32）

FRAME_NUM_REG	Bit	MCU	USB	Description	Initial State
FRAME_NUM1	[7:0]	R	W	Frame number lower byte value	00

图9-32

FRAME_NUM_REG：帧计数器（高位资料）（见图9-33）

FRAME_NUM_REG	Bit	MCU	USB	Description	Initial State
FRAME_NUM2	[7:0]	R	W	Frame number higher byte value	00

图9-33

INDEX_REG：索引寄存器（见图9-34）

INDEX：指向 USB 控制器内的某个Endpoint

INDEX_REG	Bit	MCU	USB	Description	Initial State
INDEX	[7:0]	R/W	R	Indicate a certain endpoint	00

图9-34

EP0_CSR： **Endpoint0** 控制状态寄存器（图9-35）

OUT_PKT_RDY ：当有效 OUT 通讯包由 USB 控制器写入 FIFO后，该位被置 1

IN_PKT_RDY ：当有效 IN 通讯包由 USB 控制器写入 FIFO后，该位被置 1

SENT_STALL ：如果因为总线冲突而导致控制传输中断，该位将被 USB 控制器置 1

DATA_END ：当资料发送完之后，由 CPU 置该位

SETUP_END ：当控制传输完成之后，由 CPU 置该位

SEND_STALL ：如果收到无效的信令包， CPU 应该在清除 OUT_PKT_RDY 的同时置该位

SERVICED_OUT_PKT_RDY：CPU 通过置该位来清除 OUT_PKT_RDY 状态

SERVICED_SETUP_END：CPU 通过置该位来清除 SETUP_END 状态位

EP0_CSR	Bit	MCU	USB	Description	Initial State
SERVICED_SETUP_END	[7]	W	CLEAR	The MCU should write a "1" to this bit to clear SETUP_END.	0
SERVICED_OUT_PKT_RDY	[6]	W	CLEAR	The MCU should write a "1" to this bit to clear OUT_PKT_RDY.	0
SEND_STALL	[5]	R/W	CLEAR	MCU should write a "1" to this bit at the same time it clears OUT_PKT_RDY, if it decodes an invalid token. 0 = Finish the STALL condition 1 = The USB issues a STALL and shake to the current control transfer.	0
SETUP_END	[4]	R	SET	Set by the USB when a control transfer ends before DATA_END is set. When the USB sets this bit, an interrupt is generated to the MCU. When such a condition occurs, the USB flushes the FIFO and invalidates MCU access to the FIFO.	0
DATA_END	[3]	SET	CLEAR	Set by the MCU on the conditions below: 1. After loading the last packet of data into the FIFO, at the same time IN_PKT_RDY is set. 2. While it clears OUT_PKT_RDY after unloading the last packet of data. 3. For a zero length data phase.	0
SENT_STALL	[2]	CLEAR	SET	Set by the USB if a control transaction is stopped due to a protocol violation. An interrupt is generated when this bit is set. The MCU should write "0" to clear this bit.	0
IN_PKT_RDY	[1]	SET	CLEAR	Set by the MCU after writing a packet of data into EP0 FIFO. The USB clears this bit once the packet has been successfully sent to the host. An interrupt is generated when the USB clears this bit, so as the MCU to load the next packet. For a zero length data phase, the MCU sets DATA_END at the same time.	0
OUT_PKT_RDY	[0]	R	SET	Set by the USB once a valid token is written to the FIFO. An interrupt is generated when the USB sets this bit. The MCU clears this bit by writing a "1" to the SERVICED_OUT_PKT_RDY bit.	0

图9-35

IN_CSR1_REG： **Endpoint IN 控制状态寄存器**（见图9-36）

IN_PKT_RDY ：当 CPU 将资料写入 FIFO 后，由 CPU 置该位

UNDER_RUN ：仅对 ISO 传输模式有效（当 ISO 传输时，资料来不及传输的情况）

FIFO_FLUSH ：清除 FIFO 内容控制位元

SEND_STALL ：

0： CPU 清除该位来结束 STALL 状态

1： CPU 发送一个 STALL 握手信号给 USB 控制器

SENT_STALL ： 当收到一个 STALL 的 IN 信令后，由 USB 控制器置该位

CLR_DATA_TOGGLE：切换 DATA0 和 DATA1 资料包

IN_CSR1_REG	Bit	MCU	USB	Description	Initial State
Reserved	[7]	–	–	–	0
CLR_DATA_TOGGLE	[6]	R/W	R/ CLEAR	Used in Set-up procedure. 0: There are alternation of DATA0 and DATA1 1: The data toggle bit is cleared and PID in packet will maintain DATA0	0
SENT_STALL	[5]	R/ CLEAR	SET	Set by the USB when an IN token issues a STALL handshake, after the MCU sets SEND_STALL bit to start STALL handshaking. When the USB issues a STALL handshake, IN_PKT_RDY is cleared	0
SEND_STALL	[4]	W/R	R	0: The MCU clears this bit to finish the STALL condition. 1: The MCU issues a STALL handshake to the USB.	0
FIFO_FLUSH	[3]	R/W	CLEAR	Set by the MCU if it intends to flush the packet in Input-related FIFO. This bit is cleared by the USB when the FIFO is flushed. The MCU is interrupted when this happens. If a token is in process, the USB waits until the transmission is complete before FIFO flushing. If two packets are loaded into the FIFO, only first packet (The packet is intended to be sent to the host) is flushed, and the corresponding IN_PKT_RDY bit is cleared	0
UNDER_RUN	[2]	R/ CLEAR	Set	*Valid only For Iso mode.* Set by the USB when in ISO mode, an IN token is received and the IN_PKT_RDY bit is not set. The USB sends a zero length data packet for such conditions, and the next packet that is loaded into the FIFO is flushed. This bit is cleared by writing 0.	0
Reserved	[1]	–	–	–	0
IN_PKT_RDY	[0]	R/SET	CLEAR	Set by the MCU after writing a packet of data into the FIFO. The USB clears this bit once the packet has been successfully sent to the host. An interrupt is generated when the USB clears this bit, so the MCU can load the next packet. While this bit is set, the MCU will not be able to write to the FIFO. If the MCU sets SEND STALL bit, this bit cannot be set.	0

图9-36

IN_CSR2_REG： **Endpoint IN 控制状态寄存器**（见图9-37）

IN_DMA_INT_EN：DMA中断使能位

MODE_IN ：配置响应的 Endpoint的类型（IN 还是 OUT）

ISO ：配置响应 Endpoint 的传输类型

AUTO_SET ：使能当资料量过大时，是否自动拆包

IN_CSR2_REG	Bit	MCU	USB	Description	Initial State
AUTO_SET	[7]	R/W	R	If set, whenever the MCU writes MAXP data, IN_PKT_RDY will automatically be set by the core without any intervention from MCU. If the MCU writes less than MAXP data, IN_PKT_RDY bit has to be set by the MCU.	0
ISO	[6]	R/W	R	*Used only for endpoints whose transfer type is programmable.* 1: Reserved 0: Configures endpoint to Bulk mode	0
MODE_IN	[5]	R/W	R	*Used only for endpoints whose direction is programmable.* 1: Configures Endpoint Direction as IN 0: Configures Endpoint Direction as OUT	1
IN_DMA_INT_EN	[4]	R/W	R	Determine whether the interrupt should be issued or not, when the EP1 IN_PKT_RDY condition happens. This is only useful for DMA mode. 0 = Interrupt enable, 1 = Interrupt Disable	0
Reserved	[3:0]	–	–	–	–

图7-37

OUT_CSR1_REG 和 **OUT_CSR2_REG** 的定义类同 **OUT_CSR1_REG** 等（见图9-38,9-39）

OUT_CSR1_REG	Bit	MCU	USB	Description	Initial State
CLR_DATA_ TOGGLE	[7]	R/W	CLEAR	When the MCU writes a 1 to this bit, the data toggle sequence bit is reset to DATA0.	0
SENT_STALL	[6]	R/ CLEAR	SET	Set by the USB when an OUT token is ended with a STALL handshake. The USB issues a stall handshake to the host if it sends more than MAXP data for the OUT TOKEN.	0
SEND_STALL	[5]	R/W	R	0: The MCU clears this bit to end the STALL condition handshake, IN PKT RDY is cleared. 1: The MCU issues a STALL handshake to the USB. The MCU clears this bit to end the STALL condition handshake, IN PKT RDY is cleared.	0
FIFO_FLUSH	[4]	R/W	CLEAR	The MCU writes a 1 to flush the FIFO. This bit can be set only when OUT_PKT_RDY (D0) is set. The packet due to be unloaded by the MCU will be flushed.	0
DATA_ERROR	[3]	R	R/W	*Valid only in ISO mode.* This bit should be sampled with OUT_PKT_RDY. When set, it indicates the data packet due to be unloaded by the MCU has an error (either bit stuffing or CRC). If two packets are loaded into the FIFO, and the second packet has an error, then this bit gets set only after the first packet is unloaded. This bit is automatically cleared when OUT_PKT_RDY gets cleared.	0
OVER_RUN	[2]	R/ CLEAR	R/W	*Valid only in ISO mode.* This bit is set if the core is not able to load an OUT ISO token into the FIFO. The MCU clears this bit by writing 0.	0
Reserved	[1]	–	–	–	0
OUT_PKT_RDY	[0]	R/ CLEAR	SET	Set by the USB after it has loaded a packet of data into the FIFO. Once the MCU reads the packet from FIFO, this bit should be cleared by MCU (write a "0").	0

图9-38

OUT_CSR2_REG	Bit	MCU	USB	Description	Initial State
AUTO_CLR	[7]	R/W	R	If the MCU is set, whenever the MCU reads data from the OUT FIFO, OUT_PKT_RDY will automatically be cleared by the logic without any intervention from the MCU.	0
ISO	[6]	R/W	R	Determine endpoint transfer type. 0: Configures endpoint to Bulk mode. 1: Reserved	0
OUT_DMA_INT_ MASK	[5]	R/W	R	Determine whether the interrupt should be issued or not. OUT_PKT_RDY condition happens. This is only useful for DMA mode 0 = Interrupt Enable 1 = Interrupt Disable	0

图9-39

EPn_FIFO：**FIFI**资料寄存器（见图9-40）

EPn_FIFO	Bit	MCU	USB	Description	Initial State
FIFO_DATA	[7:0]	R/W	R/W	FIFO data value	0xXX

图9-40

MAXP_REG：最大资料包长度配置寄存器（见图9-41）

MAXP_REG	Bit	MCU	USB	Description	Initial State
MAXP	[3:0]	R/W	R	0000: Reserved 0001: MAXP = 8 Byte0010: MAXP = 16 Byte 0100: MAXP = 32 Byte1000: MAXP = 64 Byte For EP0, MAXP=8 is recommended. For EP1~4, MAXP=32 or MAXP=64 is recommended. And, if MAXP=32, the dual packet mode will be enabled automatically.	0001

图9-41

OUT_FIFO_CON1_REG和**OUT_FIFO_CON1_REG**：指明 **OUT FIFO** 中有多少 **Byte** 资料（见图9-42，9-43）

OUT_FIFO_CNT1_REG	Bit	MCU	USB	Description	Initial State
OUT_CNT_LOW	[7:0]	R	W	Lower byte of write count	0x00

图9-42

OUT_FIFO_CNT2_REG	Bit	MCU	USB	Description	Initial State
OUT_CNT_HIGH	[7:0]	R	W	Higher byte of write count. The OUT_CNT_HIGH may be always 0 normally.	0x00

图9-43

EPn_DMA_CON：**Endpoint DMA** 控制器（见图9-44）

DMA_MODE_EN：使能 DMA 工作模式
IN_DMA_RUN：IN DMA 启动控制位元
OUT_DMA_RUN：OUT DMA 启动控制位元
DEMAND_MODE：DMA Demand 模式使能位元
STATE：DMA 状态标志位
IN_RUN_OB：IN DMA 状态位

EPn_DMA_CON	Bit	MCU	USB	Description	Initial State
IN_RUN_OB	[7]	R/W	W	Read) IN_DMA_Run Observation 0: DMA is stopped 1:DMA is running Write) Ignore EPn_DMA_TTC_n register 0: DMA requests will be stopped if EPn_DMA_TTC_n reaches 0. 1: DMA requests will be continued although EPn_DMA_TTC_n reaches 0.	0
STATE	[6:4]	R	W	DMA State Monitoring	0
DEMAND_MODE	[3]	R/W	R	DMA Demand mode enable bit 0: Demand mode disable 1: Demand mode enable	0
OUT_RUN_OB/ OUT_DMA_RUN	[2]	R/W	R/W	Functionally separated into write and read operation. Write operation: '0' = Stop '1' = Run Read operation: OUT DMA Run Observation	0
IN_DMA_RUN	[1]	R/W	R	Start DMA operation. 0 = Stop 1 = Run	0
DMA_MODE_EN	[0]	R/W	R/ CLEAR	Set DMA mode.If the IN_RUN_OB has been wrtten as 0 and EPn_DMA_TTC_n reaches 0, DMA_MODE_EN bit will be cleared by USB. 0 = Interrupt mode 1 = DMA mode	0

图9-44

EP0_UNIT_CNT：**Endpoint 0 DMA** 传输长度寄存器（见图9-45）

DMA_UNIT	Bit	MCU	USB	Description	Initial State
EPn_UNIT_CNT	[7:0]	R/W	R	EP DMA transfer unit counter value	0x00

图9-45

EPn_UNIT_CNT：**Endpoint DMA** 传输长度寄存器（见图9-46）

DMA_FIFO	Bit	MCU	USB	Description	Initial State
EPn_FIFO_CNT	[7:0]	R/W	R	EP DMA transfer FIFO counter value	0x00

图9-46

EPn_TTC_x：**Endpoint DMA** 总传输长度寄存器（见图9-47）

DMA_TX	Bit	MCU	USB	Description	Initial State
EPn_TTC_L	[7:0]	R/W	R	DMA total transfer count value (lower byte)	0x00
EPn_TTC_M	[7:0]	R/W	R	DMA total transfer count value (middle byte)	0x00
EPn_TTC_H	[3:0]	R/W	R	DMA total transfer count value (higher byte)	0x00

图9-47

第十章 Flash-ROM 的基本知识及其编程

一、闪存简介

Flash-ROM（闪存）已经成为了目前最成功、流行的一种固态内存，与 EEPROM 相比具有读写速度快，而与 SRAM 相比具有非易失、以及价廉等优势。而基于 NOR 和 NAND 结构的闪存是现在市场上两种主要的非易失闪存技术。 Intel 于 1988 年首先开发出 NOR flash 技术，彻底改变了原先由 EPROM 和 EEPROM 一统天下的局面。紧接着，1989 年东芝公司发表了 NAND flash 技术（后将该技术无偿转让给韩国 Samsung 公司），强调降低每比特的成本，更高的性能，并且象磁盘一样可以通过接口轻松升级。

但是经过了十多年之后，仍然有相当多的工程师分不清 NOR 和 NAND 闪存，也搞不清楚 NAND 闪存技术相对于 NOR 技术的优越之处，因为大多数情况下闪存只是用来存储少量的代码，这时 NOR 闪存更适合一些。而 NAND 则是高资料存储密度的理想解决方案。

NOR 的特点是芯片内执行（XIP，eXecute In Place），这样应用程序可以直接在闪存内运行，不必再把代码读到系统 RAM 中。NOR 的传输效率很高，在 1～4MB 的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。

NAND 结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快，这也是为何所有的 U 盘都使用 NAND 闪存做为存储介质的原因。应用 NAND 的困难在于闪存和需要特殊的系统接口。

二、性能比较

闪存是非易失内存，可以对称为块的内存单元块进行擦写和再编程。任何闪存器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除。NAND 器件执行擦除操作是十分简单的，而 NOR 则要求在进行擦除前先要将目标块内所有的位都写为 0。

由于擦除 NOR 器件时是以 64～128KB 的块进行的，执行一个写入/擦除操作的时间为5s，与此相反，擦除 NAND 器件是以 8～32KB 的块进行的，执行相同的操作最多只需要 4ms。

执行擦除时块尺寸的不同进一步拉大了 NOR 和 NAND 之间的性能差距，统计表明，对于给定的一套写入操作（尤其是更新小文件时），更多的擦除操作必须在基于 NOR 的单元中进行。这样，当选择存储解决方案时，设计师必须权衡以下的各项因素。

1) NOR的读速度比NAND稍快一些。
2) NAND的写入速度比NOR快很多。
3) NAND的4ms擦除速度远比NOR的5s快。大多数写入操作需要先进行擦除操作。
4) AND的擦除单元更小，相应的擦除电路更少。

三、接口差别

NOR 闪存带有SRAM接口，有足够的地址引脚来寻址，可以很容易地存取其内部的每一个字节。

NAND闪存使用复杂的I/O口来串行地存取资料，各个产品或厂商的方法可能各不相同。8个引脚用来传送控制、地址和资料信息。

NAND读和写操作采用512字节的块，这一点有点像硬盘管理此类操作，很自然地，基

于NAND的闪存就可以取代硬盘或其它块设备。

四、容量和成本

NAND 闪存的单元尺寸几乎是 NOR 闪存的一半，由于生产过程更为简单，NAND 结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。

NOR 闪存容量为 1～11～16MB 闪存市场的大部分，而 NAND 闪存只是用在 8MB 以上的产品当中，这也说明 NOR 主要应用在代码存储介质中，NAND 适合于资料存储，NAND 在 CompactFlash、Secure Digital、PC Cards 和 MMC 存储卡市场上所占份额最大。

五、可靠性和耐用性

采用闪存介质时一个需要重点考虑的问题是可靠性。对于需要扩展 MTBF 的系统来说，闪存是非常合适的存储方案。可以从寿命（耐用性）、位交换和坏块处理三个方面来比较 NOR 和 NAND 的可靠性。

寿命（耐用性）

在 NAND 闪存中每个块的最大擦写次数是一百万次，而 NOR 的擦写次数是十万次。NAND 内存除了具有 10：1 的块擦除周期优势，典型的 NAND 块尺寸要比 NOR 器件小 8 倍，每个 NAND 内存块在给定的时间内的删除次数要少一些。

位交换

所有闪存器件都受位交换现象的困扰。在某些情况下（很少见，NAND 发生的次数要比NOR 多），一个比特位会发生反转或被报告反转了。

一位的变化可能不很明显，但是如果发生在一个关键文件上，这个小小的故障可能导致系统停机。如果只是报告有问题，多读几次就可能解决了。

当然，如果这个位真的改变了，就必须采用错误探测/错误纠正（EDC/ECC）算法。位反转的问题更多见于 NAND 闪存，NAND 的供货商建议使用 NAND 闪存的时候，同时使用 EDC/ECC 算法。

这个问题对于用 NAND 存储多媒体信息时倒不是致命的。当然，如果用本地存储设备来存储操作系统、配置文件或其它敏感信息时，必须使用 EDC/ECC 系统以确保可靠性。

坏块处理

NAND 器件中的坏块是随机分布的。以前也曾有过消除坏块的努力，但发现成品率太低，代价太高，根本不划算。

NAND 器件需要对介质进行初始化扫描以发现坏块，并将坏块标记为不可用。在已制成的器件中，如果通过可靠的方法不能进行这项处理，将导致高故障率。

六、易于使用

可以非常直接地使用基于 NOR 的闪存，可以像其它内存那样连接，并可以在上面直接运行代码。

由于需要 I/O 接口，NAND 要复杂得多。各种 NAND 器件的存取方法因厂家而异。

在使用 NAND 器件时，必须先写入驱动程序，才能继续执行其它操作。向 NAND 器件写入信息需要相当的技巧，因为设计师绝不能向坏块写入，这就意味着在 NAND 器件上自始至终都必须进行虚拟映像。

七、软件支持

当讨论软件支持的时候，应该区别基本的读/写/擦操作和高一级的用于磁盘仿真和闪存管理算法的软件，包括性能优化。

在 NOR 器件上运行代码不需要任何的软件支持，在 NAND 器件上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序（MTD），NAND 和 NOR 器件在进行写入和擦除操作时都需要 MTD。

八、典型的 NOR 闪存（Strata Flash）

Strata Flash 是 Intel 公司产的典型 Nor Flash， 本机使用的Strata Flash 是该系列中的 28F320J3，该闪存的内部逻辑框图如图10-1：

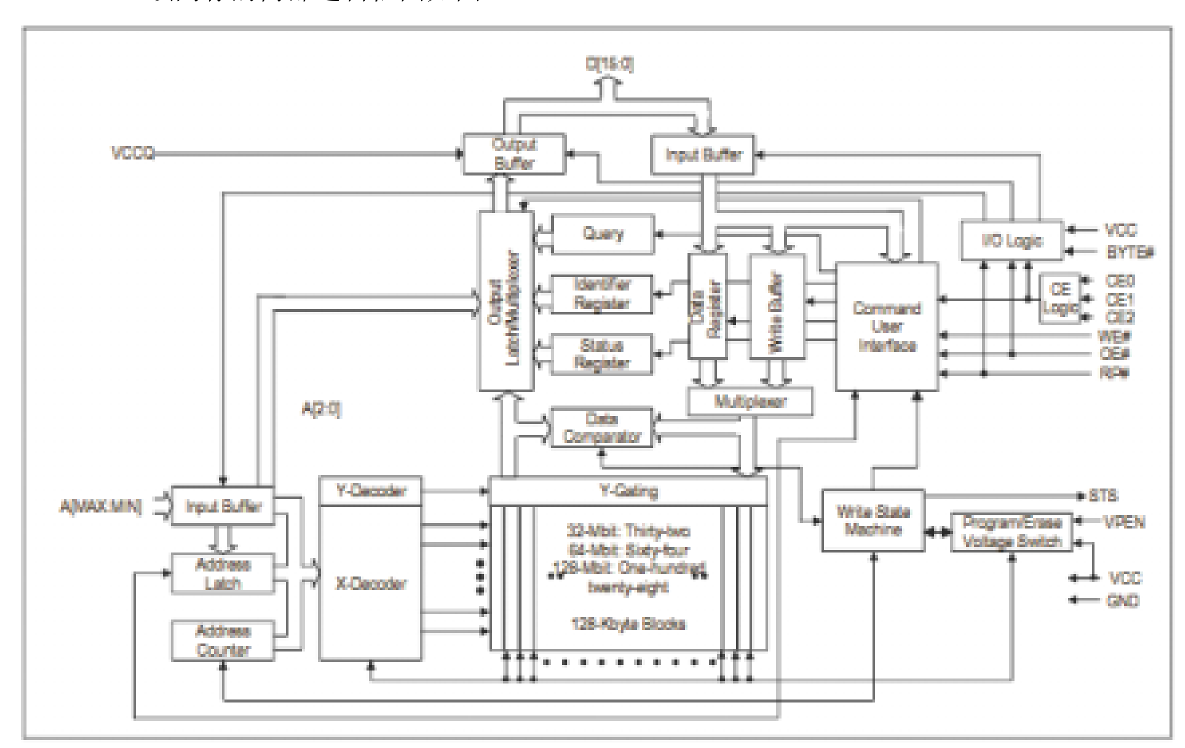


图10-1

它的特性如下：

1) 问速度有 110ns/120ns 和150ns 共 3 檔
2) 具备128bit 加密寄存器
3) 块尺寸：128KB

九、典型的 NAND 闪存（K9S5608）

K9S5608 是韩国 Samsung 公司所产的 256MBit（32MByte） SMC 卡（外形封装成卡片形式的） NAND 闪存。下图是K9S5608的内部逻辑框图。

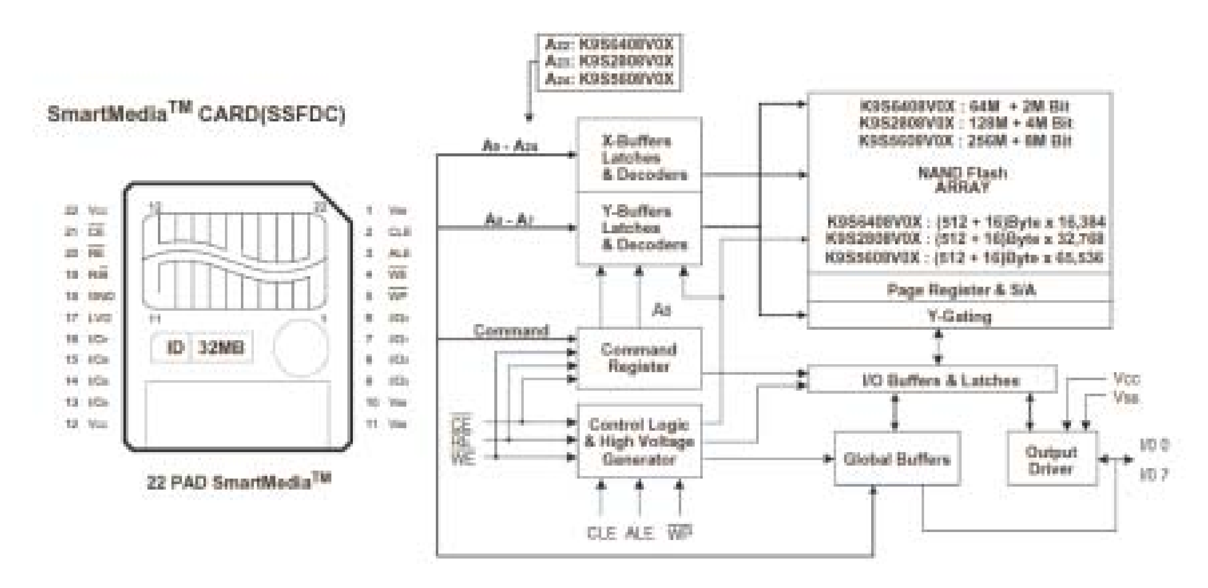


图10-2

K9S5608 具有以下特性：

（1）　　32MByte 存储空间的结构为：（32M＋1024K）bit×8bit（见图10-3）

（2）　　支持自动编程和擦除模式

（3）　　10uS 随即页面读写

（4）　　200uS 快速页面擦除周期

（5）　　具备硬件写保护功能

（6）　　擦/写寿命：10 万次

（7）　　资料保存寿命：10 年

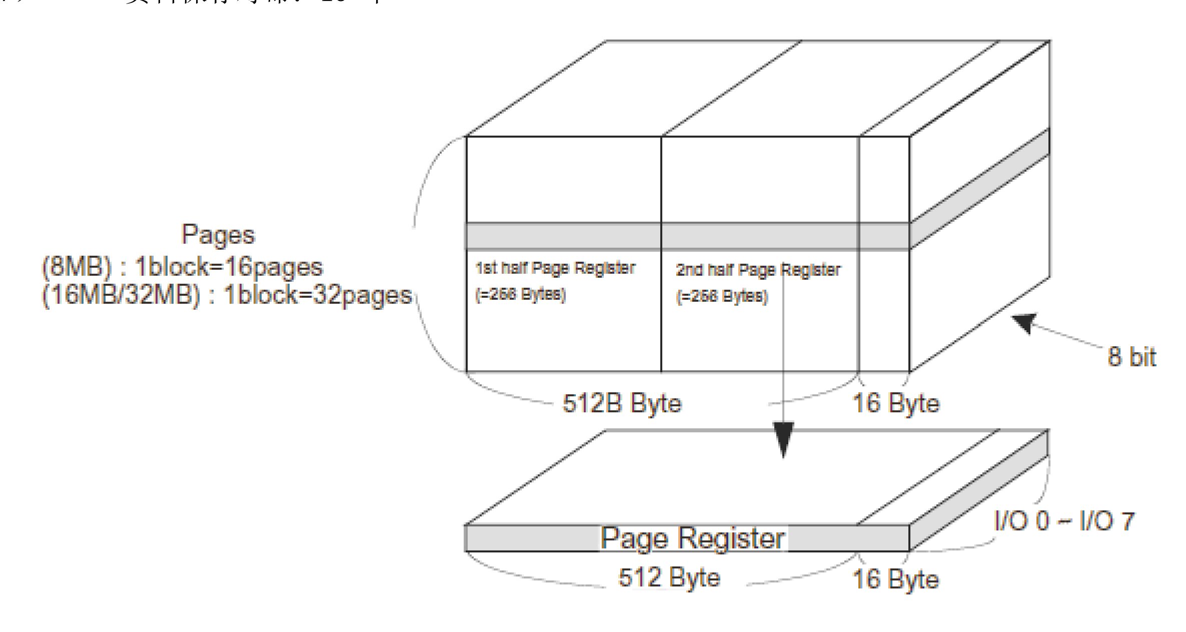


图 10-3

第十一章 CPLD 在设计中的应用

一、CPLD 概述

CPLD 是复杂可编程逻辑器件（Complex Programable Logic Device）的简称，亦是 PLD 的一种。由于采用连续连接结构。这种结构易于预测延时，从而电路仿真更加准确。CPLD 是标准的大规模集成电路产品，可用于各种数字逻辑系统的设计。近年来，由于采用先进的集成工艺和大批量生产，CPLD器件成本不断下降，集成密度、速度和性能大幅度提高，一个芯片就可以实现一个复杂的数字电路系统。CPLD 如同一张白纸或是一堆积木，工程师可以通过传统的原理图输入法，或是硬件描述语言自由的设计一个数字系统。通过软件仿真，我们可以事先验证设计的正确性。在PCB完成以后，还可以利用 CPLD 的在线修改能力，随时修改设计而不必改动硬件电路。使用 CPLD 来开发数字电路，可以大大缩短设计时间，减少 PCB 面积，提高系统的可靠性。 CPLD 的这些优点使得 CPLD 技术在 90 年代以后得到飞速的发展，同时也大大推动了 EDA 软件和硬件描述语言（HDL）的进步。

大部分 CPLD 的内部逻辑结构是基于乘积项的。采用这种结构的 CPLD 芯片有：Altera 的MAX7000， MAX3000 系列（EEPROM 工艺），Xilinx 的 XC9500 系列（ Flash 工艺）和 Lattice，Cypress 的大部分产品（ EEPROM工艺）。

二、CPLD 原理

我们先看一下这种CPLD的总体结构（以 MAX7000 为例，其它型号的结构与此都非常相似）：

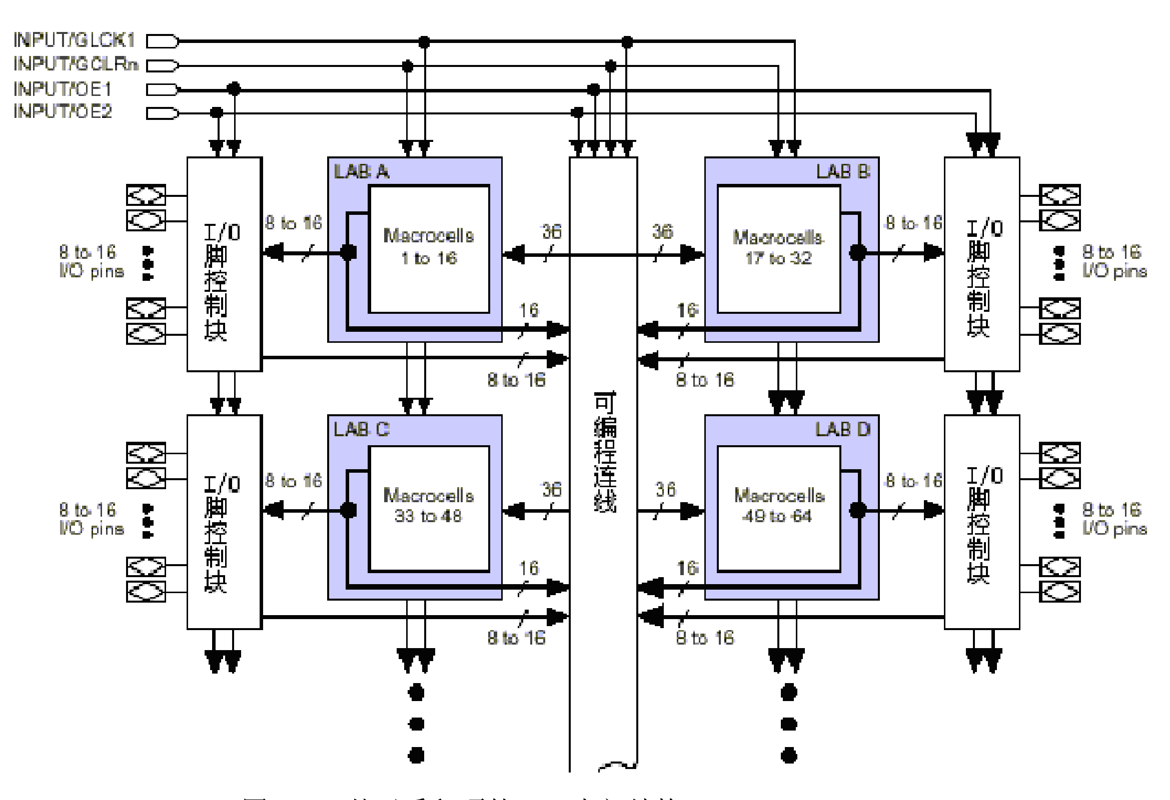


图 11-1 基于乘积项的 PLD 内部结构

这种 CPLD 可分为三块结构：宏单元（Marocell），可编程联机（PIA） 和 I/O 控制块。 宏单元是 CPLD 的基本结构，由它来实现基本的逻辑功能。图 11-1 中 LABx 部分是多

个宏单元的集合（因为宏单元较多，没有一一画出）。可编程联机负责信号传递，连接所有的宏单元。I/O 控制块负责输入输出的电气特性控制，比如可以设定集电极开路输出，摆率控制，三态输出等。图 11-1 左上角的 INPUT/GCLK1，INPUT/GCLRn，INPUT/OE1，INPUT/OE2 是全局时钟，清零和输出使能信号，这几个信号有专用联机与 PLD 中每个宏单元相连，信号到每个宏单元的延时相同并且延时最短。

宏单元的具体结构见下图（图 11-2）：

图 11-2 宏单元结构

左侧是乘积项数组，实际就是一个与或数组，每一个交叉点都是一个可编程熔丝，如果导通就是实现“与”逻辑。后面的乘积项选择矩阵是一个“或”数组。两者一起完成组合逻辑。图右侧是一个可编程 D 触发器，它的时钟，清零输入都可以编程选择，可以使用专用的全局清零和全局时钟，也可以使用内部逻辑（乘积项数组）产生的时钟和清零。如果不需要触发器，也可以将此触发器旁路，信号直接输给 PIA 或输出到 I/O 脚。

下面我们以一个简单的电路为例,具体说明 PLD 是如何利用以上结构实现逻辑的，电路如下图(图 11-3)：

图 11-3

假设组合逻辑的输出(AND3 的输出)为 f，则 f=(A+B)*C*(!D)=A*C*!D + B*C*!D (我们以!D 表示 D 的“非”)。PLD 将以下面的方式来实现组合逻辑 f:

图 11-4

A，B，C，D 由 PLD 芯片的管脚输入后进入可编程联机数组（ PIA ），在内部会产生 A，A 反，B，B 反，C，C 反，D，D 反共 8 个输出。图中每一个叉表示相连（可编程熔丝导通），所以得到：f= f1 + f2 = (A*C*!D) + (B*C*!D) 。这样组合逻辑就实现了。 图 11-4 电路中 D 触发器的实现比较简单，直接利用宏单元中的可编程 D 触发器来实现。时钟信号 CLK 由 I/O 脚输入后进入芯片内部的全局时钟专用信道，直接连接到可编程触发器的时钟端。可编程触发器的输出与 I/O 脚相连，把结果输出到芯片管脚。这样 PLD 就完成了图 9-3 所示电路的功能。（以上这些步骤都是由软件自动完成的，不需要人为干预）。

图 11-4 的电路是一个很简单的例子，只需要一个宏单元就可以完成。但对于一个复杂的电路，一个宏单元是不能实现的，这时就需要通过并联扩展项和共享扩展项将多个宏单元相连，宏单元的输出也可以连接到可编程联机数组，再做为另一个宏单元的输入。这样 PLD 就可以实现更复杂逻辑。

这种基于乘积项的 PLD 基本都是由 EEPROM 和 Flash 工艺制造的，一上电就可以工作，无需其它芯片配合。

三、MAX7000A 简介

MAX7000A 系列 CPLD 是美国 Altera 公司在原 MAX7000S等产品基础上推出的高性能CPLD。它具有以下特点：

（1）3．3V 工作电压，并支持 ISP（在系统可编程）

（2）内置符合 IEEE1149.1标准的边界扫描测试（BST）

（3）管脚完全兼容 5V 版本的 MAX7000S 系列

（4）Pin-to-Pin 的信号延时仅有 4.5ns，最高工作频率可高达 227.3MHz

（5）支持混合电压供电，即：内核工作在 3.3V，而 I/O 可工作在5.0V，3.3V 或 2.5V 逻辑电平

（6）支持管脚的摆率控制，以获得较佳的 I/O EMI 特性

下图是MAX7000A系列的一些主流型号的特性参照图，以及速度分文件（图11-5，11-6）

Feature	EPM7032AE	EPM7064AE	EPM7128AE	EPM7256AE	EPM7512AE
Usable gates	600	1,250	2,500	5,000	10,000
Macrocells	32	64	128	256	512
Logic array blocks	2	4	8	16	32
Maximum user I/O pins	36	68	100	164	212
t_{PD} (ns)	4.5	4.5	5.0	5.5	7.5
t_{SU} (ns)	2.9	2.8	3.3	3.9	5.6
t_{FSU} (ns)	2.5	2.5	2.5	2.5	3.0
t_{CO1} (ns)	3.0	3.1	3.4	3.5	4.7
f_{CNT} (MHz)	227.3	222.2	192.3	172.4	116.3

图11-5

Device	Speed Grade					
	-4	-5	-6	-7	-10	-12
EPM7032AE	✓			✓	✓	
EPM7064AE	✓			✓	✓	
EPM7128A			✓	✓	✓	✓
EPM7128AE		✓		✓	✓	
EPM7256A			✓	✓	✓	✓
EPM7256AE		✓		✓	✓	
EPM7512AE				✓	✓	✓

图11-6

第十二章 UC/OS-II 在 S3C2410 上的移植

整个移植工作可以分为两个方面，一部分是和 ARM 相关，一部分是和移植原理相关。在开始实际的移植工作前，需要对这两部分有一定的背景知识，尤其是和侧重于和移植工作相关的概念和原理。下面分别做一些介绍：

一、ARM 的体系结构

ARM（Advanced RISC Machines）是目前在嵌入式领域里应用最广泛的 RISC 微处理器结构，以其低成本、低功耗、高性能的特点占据了嵌入式系统应用领域的领先地位。ARM 系列的处理器当前有 ARM7、ARM9、ARM9E、ARM10 等多个产品，此外 ARM 公司合作伙伴，例如 Intel 也提供基于 XScale 微体系结构的相关处理器产品。所有的 ARM 处理器都共享 ARM 通用的基础体系结构，所以开发者在不同的 ARM 处理器上做操作系统移植时，可以将节省相当多的工作量，这无疑将大大降低软件开发成本。

要详细完整的了解 ARM 的体系结构，当然是去读 ARM Architectur Reference Manual ，这是一个 13M 的 pdf 文档，有 800 多页，可以从 ARM 的网站下载.

处理器模式： (cpu mode)

ARM 的处理器可以工作在 7 种模式，如下图所示：

ARM version 4 processor modes

Processor mode		Description
User	usr	Normal program execution mode
FIQ	fiq	Supports a high-speed data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
System	sys	Runs privileged operating system tasks (ARM architecture version 4 and above)

这里除 usr 模式以外的其它模式都叫做特权模式，除 usr 和 sys 外的其它 5 种模式叫做异常模式。在 usr 模式下对系统资源的访问是受限制的，也无法主动地改变处理器模式。异常模式通常都是和硬件相关的，例如中断或者是试图执行未定义指令等。这里需要强调的是和移植相关的两种处理器模式：svc 态和 irq 态，分别指操作系统的保护模式和通用中断处理模式。这两种模式之间的转换可以通过硬件的方式，也可以通过软件的方式。uC/OS-II 内核在执行过程中，大部分时间都是工作在 svc 态，当有硬件中断，例如时钟中断到来时，cpu 硬件上会自动完成从 svc 态进入 irq 态，在中断处理程序的结束处，则需要通过编程的方法使得 cpu 从 irq 态恢复到 svc 态，这个在移植代码中可以找到。

程序状态寄存器： (PSR：Program status register)

在任何一种处理器模式中，都使用同一个寄存器来标识当前处理器的工作模式：这个寄存器叫做 CPSR (Current Program Status Register)，它的 [0--4] 位用来表示 cpu mode：

The format of the CPSR and the SPSRs is shown below.

31	30	29	28	27	26 … 8	7	6	5	4	3	2	1	0
N	Z	C	V	Q	DNM(RAZ)	I	F	T	M4	M3	M2	M1	M0

Table 2-2 The mode bits

M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARM architecture v4 and above)

每一种处理器异常模式，都有一个对应的 SPSR (Saved Program Status Register)寄存器，用来保存进入异常模式前的 CPSR。SPSR 的作用就是当从异常模式退出时，可以通过一条简单的汇编指令就能够恢复进入异常模式前的 CPSR，而这个值都是保存在当前异常模式的 SPSR 中的。例如：当从 usr 态进入中断 irq 态时，原先的 CPSR_all 将被保存在当前的 SPSR_irq 中，类似的异常模式下的 SPSR 还有 SPSR_fiq、SPSR_svc、SPSR_abt、SPSR_und。非异常模式的 usr 和 sys 模式下没有 SPSR，只有 CPSR。不能显式的指定把 CPSR 保存到某个异常模式下的 SPSR，比如 SPSR_irq，而必须是变更到 irq 态之后 cpu 自动完成的，不能在其它态下硬性赋值，因为 SPSR_irq 是其它状态下不可见的。

ARM 寄存器：(register)

ARM 处理器一共有 37 个寄存器，其中 31 个是通用寄存器，包括一个程序计数器 PC。另外 6 个就是上面提到的程序状态寄存器。

1 通用寄存器：

a) R0－R7：与所有处理器模式无关的寄存器，可以用作任何用途。

b) R8－R14：与处理器模式有关的寄存器，在不同的模式下，对应到不同的物理寄存器。其中 R13 又叫做 sp，一般用于堆栈指针。R14 又叫做 lr，一般用于保存返回地址。这两个寄存器在每种异常模式下都对应到不同的物理寄存器上，例如 lr_irq、lr_svc、lr_fiq 等。

c)R15：又叫做程序计数器，即 pc，所有的模式下都使用同一个 pc。

2 状态寄存器：

a) CPSR：当前程序状态寄存器，所有的模式下都使用同一个 CPSR。

b) SPSR：保存的程序状态寄存器，每种异常模式下都有自己的 SPSR，一共有 5 种 SPSR，即 SPSR_irq、SPSR_fiq、SPSR_svc、SPSR_abt、SPSR_und。usr 和 sys 态下没有 SPSR 。

所有的 ARM 寄存器的命名和含义，可以用下面的这张表来说明，其中相同命名的都是同一个物理寄存器，不同命名的寄存器都对应不同的物理寄存器。

Modes						
	Privileged modes					
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

二、uC/OS-II 移植工作介绍

uC/OS-II 实际上可以简单地看作是一个多任务的调度器，在这个任务调度器之上完善并添加了和多任务操作系统相关的一些系统服务，如信号量、邮箱等。它的 90%的代码都是用 C 语言写的，因此只要有相应的 C 语言编译器，基本上就可以直接移植到特定处理器上，这也是 uC/OS-II 具有良好的可移植性的原因。移植工作的绝大部分都集中在多任务切换的实现上，因为这部分代码主要是用来保存和恢复处理器现场（即相关寄存器），因此不能用 C 语言，只能使用特定的处理器汇编语言完成。

uC/OS-II 的全部源代码量大约是 6000－7000 行，一共有 15 个文件。将 uC/OS-II 移植到 ARM 处理器上，需要完成的工作也非常简单，只需要修改三个和 ARM 体系结构相关的文件，代码量大约是 500 行。以下分别介绍这三个文件的移植工作：

1、 OS_CPU.H 文件

- 数据类型定义

这部分的修改是和所用的编译器相关的，不同的编译器会使用不同的位组长度来表示同一数据类型，比如 int，同样在 x86 平台上，如果用 GNU 的 gcc 编译器，则编译为 4 bytes，而使用 MS VC＋＋则编译为 2 bytes。

- 堆栈单位

因为处理器现场的寄存器在任务切换时都将会保存在当前运行任务的堆栈中，所以 OS_STK 数据类型应该是和处理器的寄存器长度一致的。

- 堆栈增长方向

堆栈由高地址向低位址增长，这个也是和编译器有关的，当进行函数调用时，入口参数和返回地址一般都会保存在当前任务的堆栈中，编译器的编译选项和由此生成的堆栈指令就会决定堆栈的增长方向。

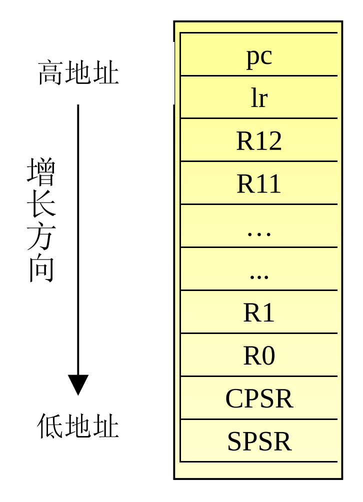
- 宏定义

包括开关中断的宏定义，以及进行任务切换的宏定义。

2、 OS_CPU_C.C 文件

- 任务堆栈初始化

这里涉及到任务初始化时的一个堆栈设计，也就是在堆栈增长方向上如何定义每个需要保存的寄存器位置，在 ARM 体系结构下，任务堆栈空间由高至低依次将保存着 pc、lr、r12、r11、r10、… r1、r0、CPSR、SPSR。



这里需要说明两点，一是当前任务堆栈初始化完成后，OSTaskStkInit 返回新的堆栈指针 stk，在 OSTaskCreate（）执行时将会调用 OSTaskStkInit 的初始化过程，然后通过 OSTCBInit（）函数调用将返回的 sp 指针保存到该任务的 TCB 块中。二是初始状态的堆栈其实是模拟了一次中断发生后的堆栈结构，因为任务被创建后并不是直接就获得执行的，而是通过 OSSched（）函数进行调度分配，满足执行条件后才能获得执行的。为了使这个调度简单一致，就预先将该任务的 pc 指针和返回地址 lr 都指向函数入口，以便被调度时从堆栈中恢复刚开始运行时的处理器现场。

- 系统 hook 函数

此外，在这个文件里面还需要实现几个操作系统规定的 hook 函数，如下：

OSSTaskCreateHook( )

OSTaskDelHook( )

OSTaskSwHook( )

OSTaskStatHook( )

OSTimeTickHook( )

如果没有特殊需求，则只需要简单地将它们都实现为空函数就可以了。

3、 OS_CPU_A.S 文件

- **OSStartHighRdy（）**

此函数是在 OSStart（）多任务启动之后，负责从最高优先级任务的 TCB 控制块中获得该任务的堆栈指针 sp，通过 sp 依次将 cpu 现场恢复，这时系统就将控制权交给用户创建的该任务进程，直到该任务被阻塞或者被其它更高优先级的任务抢占 cpu。该函数仅仅在多任务启动时被执行一次，用来启动第一个，也就是最高优先级的任务执行，之后多任务的调度和切换就是由下面的函数来实现。

- **OSCtxSw（）**

任务级的上下文切换，它是当任务因为被阻塞而主动请求 cpu 调度时被执行，由于此时的任务切换都是在非异常模式下进行的，因此区别于中断级别的任务切换。它的工作是先将当前任务的 cpu 现场保存到该任务堆栈中，然后获得最高优先级任务的堆栈指针，从该堆栈中恢复此任务的 cpu 现场，使之继续执行。这样就完成了一次任务切换。

- **OSIntCtxSw（）**

中断级的任务切换，它是在时钟中断 ISR（中断服务例程）中发现有高优先级任务等待的时钟信号到来，则需要在中断退出后并不返回被中断任务，而是直接调度就绪的高优先级任务执行。这样做的目的主要是能够尽快地让高优先级的任务得到响应，保证系统的实时性能。它的原理基本上与任务级的切换相同，但是由于进入中断时已经保存过了被中断任务的 cpu 现场，因此这里就不用再进行类似的操作，只需要对堆栈指针做相应的调整，原因是函数的嵌套。

- **OSTickISR（）**

时钟中断处理函数，它的主要任务是负责处理时钟中断，调用系统实现的 OSTimeTick 函数，如果有等待时钟信号的高优先级任务，则需要在中断级别上调度其执行。其它相关的两个函数是 OSIntEnter（）和 OSIntExit（），都需要在 ISR 中执行。

- **ARMEnableInt（）& ARMDisableInt（）**

分别是退出临界区和进入临界区的宏指令实现。主要用于在进入临界区之前关闭中断，在退出临界区的时候恢复原来的中断状态。它的实现比较简单，可以采用方法 1 直接开关中断来实现，也可以采用方法 2 通过保存关闭/恢复中断屏蔽位来实现。

移植 uC/OS-II 的绝大部分工作都集中在 os_cpu_a.S 文件的移植，这个文件的实现集中体现了所要移植到处理器的体系结构和 uC/OS-II 的移植原理；在这个文件里，最困难的工作又集中体现在 OSIntCtxSw 和 OSTickISR 这两个函数的实现上。这是因为这两个函数的实现是和移植者的移植思路以及相关硬件定时器、中断寄存器的设置有关。在实际的移植工作中，这两个地方也是比较容易出错的地方。

OSIntCtxSw 最重要的作用就是它完成了在中断 ISR 中直接进行任务切换，从而提高了实时响应的速度。它发生的时机是在 ISR 执行到 OSIntExit 时，如果发现有高优先级的任务因为等待的 time tick 到来获得了执行的条件，这样就可以马上被调度执行，而不用返回被中断的那个任务之后再进行任务切换，因为那样的话就不够实时了。

实现 OSIntCtxSw 的方法大致也有两种情况：一种是通过调整 sp 堆栈指针的方法，根据所用的编译器对于函数嵌套的处理，通过精确计算出所需要调整的 sp 位置来使得进入中断时所作的保存现场的工作可以被重用。这种方法的好处是直接在函数嵌套内部发生任务切换，使得高优先级的任务能够最快的被调度执行。但是这个办法需要和具体的编译器以及编译参数的设置相关，需要较多技巧。

另一种是设置需要切换标志位元的方法，在 OSIntCtxSw 里面不发生切换，而是设置一个需要切换的标志，等函数嵌套从进入 OSIntExit => OS_ENTER_CRITICAL() =>

OSIntCtxSw() => OS_EXIT_CRITICAL() => OSIntExit 退出后，再根据标志位来判断是否需要进行中断级的任务切换。这种方法的好处是不需要考虑编译器的因素，也不用做计算，但是从实时响应上不是最快，不过刚开始学习这种方法比较容易理解，实现起来也简单。

在中断态下进行任务切换，需要特别说明的一个问题是如何获得被中断任务的 lr_svc 。因为进入中断态后，lr 变成了 lr_irq ，原来任务的 lr_svc 无法在中断态下获得，这样要得到 lr_svc ，就必须在中断 ISR 里面进行一次 cpu mode 强制转换，即对 CPSR 赋值为 0x000000d3 ，只有返回到 svc 态之后才能得到 原来任务的 lr ，这个对于任务切换很重要。还有一个需要留意的问题是在强制 CPSR 变成 svc 态之后，SPSR 也会相应地变成 SPSR_irq ，这样就需要在强制转变之前保存 SPSR ，也就是被中断任务中断前的 CPSR 。具体实现请参考移植代码。

第十三章 UC/OS-II 在 S3C2410 上中断程序的编写

一、编写原理

μＣ／ＯＳ－ＩＩ包括中断管理、任务管理、时间管理、任务之间通信管理和内存管理五方面功能。其结构共分三层，如图１。Ｉ层为与处理器相关的代码，在μＣ／ＯＳ－ＩＩ的Ｉｎｔｅｌ ８０ｘ８６版本上为ＯＳ＿ＣＰＵ．Ｈ、ＯＳ＿ＣＰＵ＿Ｃ．Ｃ和ＯＳ＿ＣＰＵ＿Ａ．ＡＳＭ三个文件。该层完成系统时钟的设置、出入中断的管理和任务切换功能，为第ＩＩ层提供接口。ＩＩ层包括时间管理、任务调度管理、任务间的通信管理和内存管理四部分，是ＯＳ的主体部分，全部由ＡＮＳＩ Ｃ代码写成，与处理器无关，它为用户应用程序提供接口。ＩＩＩ层是用户应用程序部分，μＣ／ＯＳ－ＩＩ有中断和任务两个处理级别，用户可以建立自己的任务，编写必要的中断子程，在任务之间或任务与中断子程之间建立信号量、邮箱或消息队列完成控制器软件的编写。根据以上结构特点，在移植过程中，只需将Ｉ层代码针对ＭＰＣ５５５的编程结构做相应改动，使其完成系统时钟设置、中断管理和任务切换功能即可。

在前后台系统中，提供一个ＣＰＵ堆栈。发生中断时，将当前使用到的寄存器压入堆栈，保存现场，执行中断程序；中断程序完成后，从ＣＰＵ堆栈中弹出寄存器的值，恢复现场。

在多任务系统μＣ／ＯＳ－ＩＩ中不是这样。ＯＳ创建时，为每个任务建立并初始化一个堆栈。当发生中断或任务切换时，把当前任务运行现场保存起来，即将所有寄存器保存到该“旧”任务的堆栈中。当某个任务需要从就绪状态激活到运行状态时，ＯＳ又需将所有寄存器从该“新”任务的堆栈中弹出。这样，每个任务分时占用ＣＰＵ。而对各任务来说，每次进入运行态时，ＣＰＵ状态都与上次从运行态退出时完全一样。所以不再是使用一个ＣＰＵ堆栈，而是多个任务将各自的运行现场保存到自己的堆栈中。

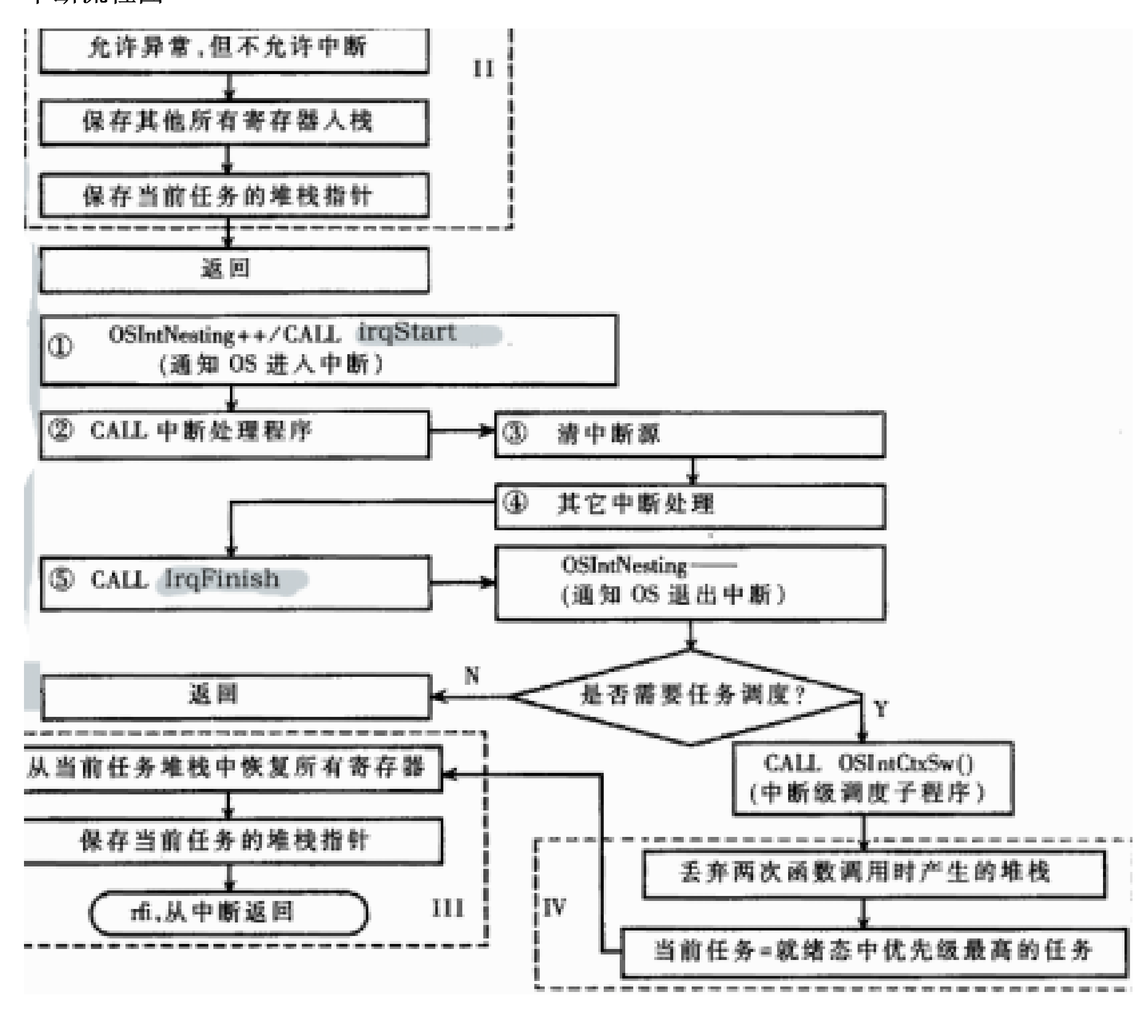
ARM 中断定义

由于某种事件的发生,而导致程序流程的改变,产生中断的事件称为中断源

CPU 响应中断的条件

至少有一个中断源给 CPU 发中断信号
系统允许中断,并且没有屏蔽此中断信号
中断类型
- a 硬件中断
- b 外部中断
- c 陷井中断
- d 现场控制量的中断

中断流程图



另外，调用Ｃ函数时也会使用到堆栈，此时编译器会创建一个堆栈；在Ｃ函数返回时，将其释放。其大小因Ｃ函数使用到的变量和编译器的不同而不同。在移植时，能够正确创建、初始化、保存并恢复各个任务的堆栈，是确保ＯＳ任务切换和中断管理顺利完成的关键。

二、 中断管理

首先，分析一下 ARM 的中断结构。在 ARM 中有新的概念——异常（Ｅｘｃｅｐｔｉｏｎ）。它包含所有ＣＰＵ非正常事件的出现，包括中断、总线错误、指令错误、系统调用异常、实时中断异常和复位等 ARM 为异常提供了异常向量表。该表为每个异常提供一个２５６字节的异常处理代码空间。

所有外部中断和Ｉ／Ｏ子模块产生的中断共同作为异常的一种，占用异常向量表中的一个位置。在该异常处理程序中，软件需根据中断状态寄存器的值判断到底发生了哪个中断并进行相应处理。

在每次发生异常时 ARM 自动将主状态寄存器 CPSR 保存到ＳPSR 中，将程序指针ＰＣ保存到 LR 中；然后ＰＣ指针指向该异常在异常向量表中的起始位置，进入异常处理程序。每次异常返回时，系统将 SPSR 中的值返回 CPSR 中，将 LR 中的值返回ＰＣ中，即程序从ＰＣ指向的位置继续执行。在发生异常和异常返回之间，不自动允许新的异常和中断。所以，程序需要在保存 CPSR 和 SPSR 后允许异常，在适当的时候允许中断。

μＣ／ＯＳ－ＩＩ的异常处理过程中，用户及ＯＳ与硬件无关的代码完成图 2 中①、②、③、④、⑤这五个步骤。依次完成以下任务：①给ＯＳＩｎｔＮｅｓｔｉｎｇ加 1 或调用 IrqStart（），通知ＯＳ，系统已进入中断；②分析中断源调用相应中断处理子程；③在该中断处理子程中完成清中断源；④进行其它中断处理；⑤调用 IrqFinish（）判断是否有更高优先级的任务被激活而需要进行任务调度，若不需要，则直接从中断返回；若需要，则调用ＯＳＩｎｔＣｔｘＳｗ（）完成中断级任务调度。在我们实现中，ＯＳＩｎｔＣｔｘＳｗ（）被汇编函数_CON_SWAP 代替。

应注意，在中断级任务调度过程中，①、⑤两处Ｃ函数被调用后不需要返回，所以需要将堆栈指针ＳＰ向下做适当调整，以丢弃这两个函数调用时编译器产生的堆栈。Ｃ函数调用时，产生堆栈的大小与编译器相关，因此应根据编译器产生的代码决定此处丢弃堆栈的大小。为保证异常时需要丢弃的堆栈大小不变，可使用图 2 中的方法，在异常处理时另外调用函数完成步骤③、④，以确保不同异常处理过程中，①、⑤两处Ｃ函数被调用时，编译器建立的堆栈大小一致。

三、任务切换

μＣ／ＯＳ－ＩＩ中的任务调度由函数ＯＳＳｃｈｅｄ（）完成。在 ARM 系统上，ＯＳＳｃｈｅｄ?穴?雪在获得当前新的最高优先级的任务指针后，调用ＣＰＵ软中断完成任务切换。

该函数完成以下三个步骤：

①将当前任务运行环境保存到当前任务的堆栈中；

②调用任务级调度函数ＯＳＣｔｘＳｗ（），将新的高优先级就绪态任务调整为当前任务；

③从新任务堆栈中弹出所有寄存器的值，恢复中断，完成任务切换。

其中①、③两部分代码与中断管理程序相同，不需要重新编写，只需编写函数ＯＳＣｔｘＳｗ（）完成任务指针的切换工作。

任务切换过程不可以被打断，所以，上述过程中始终不能打开中断。

四、驱动程序使用中断的典型例子

我们编写外部设备的驱动程序经常使用的方法为查询和中断方式.查询方式比较适合慢速设备,比如鼠标,键盘.,串口等待.对于快速设备,例如网卡,CAN 总线接口,一般使用中断方式,这样能是 CPU 得到最大程度的利用.但在中断中不能作太多的事情,否则系统的响应会很慢.一种比较合适的方法就是可以在中断中作必要的工作,然后通知应用程序,让某个应用程序去作剩下的工作.通知应用程序的方法有很多,UC/OS 中提供了消息队列,信号量,互斥量.下面我们举一个使用信号量的例子。

驱动程序初始化部分代码

{

设备初始化

信号量初始化

```
挂接中断处理程序
  }
中断处理程序
  {
进行必要的清除工作
释放信号量
  }
驱动应用程序
  {
while(1)
  {
等待信号量
读取外部设备的资料进行处理
  }
  }
```

第十四章 uC/GUI 在 ARM9 平台上的移植与应用

uC/GUI 是一个通用的嵌入式图形用户接口模块，完全用 C 语言代码组成，且向使用者提供所有的源代码，具有可移植性强、模块化程度高、可裁减性好、占用资源少等特点。另外，uC/GUI 与 uC/OSII 嵌入式操作系统一样，也是由美国 Micrium 公司开发。UC/GUI 极大地弥补了 uC/OSII 嵌入式操作系统上无 GUI 的不足，又由于出自同一公司，两者之间无缝连接，因此 uC/GUI 也成为 uC/OSII 应用上的首选图形用户接口模块。

UC/GUI 的目录结构十分清晰，大致分为以下几个目录：

目 录 名 称	说 明
Config	设置文件
GUI\AntiAlias	反色显示相关文件 （可 选）
GUI\Convertmono	单色和灰度显示的颜色转换程序文件
GUI\ConvertColor	彩色显示的颜色转换程序文件
GUI\Core	内核文件
GUI\Font	字体文件
GUI\LCDDriver	ＬＣＤ 驱动及相关文件
GUI\MemDev	内存设备相关文件 （可 选）
GUI\Touch	触摸屏驱动及相关文件 （可 选）
GUI\Widget	各个控件相关文件 （可 选）
GUI\WM	窗口管理相关文件 （可 选）

UC/GUI 在设计时各部分都被划分为独立的模块，这使得 uC/GUI 的移植工作变得相对地简单了许多。关于 uC/GUI 的移植，大致按以下几个步骤：

1.修改 uC/GUI 的 Config 目录下的设置文件。

 1） UI_Configh.h 主设置文件。

 2） GUI_TouchConf.h 触摸屏相关设置文件。

 3） LCD_Conf.h ＬＣＤ相关设置文件。

2.修改 uC/GUI 的 LcdDriver 目录下的 ＬＣＤ 驱动程序。

3.如需要支持 uC/OSII，修改 GUI_X_UCOS-II.C 文件。

4.如需要支持触摸屏，修改 GUI\Touch 目录下的相应文件。

5.如需要修改字体，修改 GUI\Font 目录下的相应文件。

UC/GUI 给用户提供了丰富的示例代码和详细的说明文文件，使得用户能够快速熟悉和应用 uC/GUI 在他们的产品上。另外它还提供了一个在 PC 机上的仿真器，通过该仿真器，用户可以十分方便地在 ＰＣ 机上调试他们的接口应用程序，程序调试好后，不需要任何改动，即可转移到硬件开发平台上运行。

第十五章　嵌入式 LINUX 开发

嵌入式 Linux 开发大致涉及三个层次：引导装载程序、Linux 内核和图形用户接口（或称 GUI）。

一、引导装载程序

引导装载程序通常是在任何硬件上执行的第一段代码。在象台式机这样的常规系统中，通常将引导装载程序装入主引导记录（Master Boot Record，(MBR)）中，或者装入 Linux 驻留的磁盘的第一个扇区中。通常，在台式机或其它系统上，BIOS 将控制移交给引导装载程序。这就提出了一个有趣的问题：谁将引导装载程序装入（在大多数情况中）没有 BIOS 的嵌入式设备上呢？

解决这个问题有两种常规技术：专用软件和微小的引导代码（tiny bootcode）。

专用软件可以直接与远程系统上的闪存设备进行交互并将引导装载程序安装在闪存的给定位置中。**闪存设备**是与存储设备功能类似的特殊芯片，而且它们能持久存储信息 — 即，在重新引导时不会擦除其内容。

这个软件使用目标（在嵌入式开发中，嵌入式设备通常被称为**目标**）上的 JTAG 端口，它是用于执行外部输入（通常来自主机机器）的指令的接口。JFlash-linux 是一种用于直接写闪存的流行工具。它支持为数众多的闪存芯片；它在主机机器（通常是 i386 机器 — 本文中我们把一台 i386 机器称为**主机**）上执行并通过 JTAG 接口使用并行端口访问目标的闪存芯片。当然，这意味着目标需要有一个并行接口使它能与主机通信。Jflash-linux 在 Linux 和 Windows 版本中都可使用，可以在命令行中用以下命令启动它：

```
Jflash-linux <bootloader>
```

某些种类的嵌入式设备具有**微小的引导代码** — 根据几个位元组的指令 — 它将初始化一些 DRAM 设置并启用目标上的一个串行（或者 USB，或者以太网）端口与主机程序通信。然后，主机程序或装入程序可以使用这个连接将引导装载程序传送到目标上，并将它写入闪存。

在安装它并给予其控制后，这个引导装载程序执行下列各类功能：

- 初始化 CPU 速度
- 初始化内存，包括启用内存库、初始化内存配置寄存器等
- 初始化串行端口（如果在目标上有的话）
- 启用指令 / 数据高速缓存
- 设置堆栈指针
- 设置参数区域并构造参数结构和标记（这是重要的一步，因为内核在标识根设备、页面大小、内存大小以及更多内容时要使用引导参数）
- 执行 POST（加电自检）来标识存在的设备并报告任何问题
- 为电源管理提供挂起 / 恢复支持
- 跳转到内核的开始

带有引导装载程序、参数结构、内核和文件系统的系统典型内存布局可能如下所示：

清单 1. 典型内存布局

```
/* Top Of Memory */

    Bootloader
    Parameter Area
    Kernel
    Filesystem

 /* End Of Memory */
```

嵌入式设备上一些流行的并可免费使用的 Linux 引导装载程序有 Blob、Redboot 和 Bootldr（请参阅参考资料获得链接）。所有这些引导装载程序都用于基于 ARM 设备上的 Linux，并需要 Jflash-linux 工具用于安装。,我们所使用的引导装载程序为 VIVI

一旦将引导装载程序安装到目标的闪存就会执行我们上面提到的所有初始化工作。然后，它准备接收来自主机的内核和文件系统。一旦装入了内核，引导装载程序就将控制转给内核。

二、设置工具链

设置工具链在主机机器上创建一个用于编译将在目标上运行的内核和应用程序的构建环境 — 这是因为目标硬件可能没有与主机兼容的二进制执行级别。

工具链由一套用于编译、汇编和链接内核及应用程序的组件组成。 这些组件包括：

Binutils 用于操作二进制文件的实用程序集合。它们包括诸如 ar、as、objdumpobjcopy 这样的实用程序。

Gcc GNU C 编译器。

Glibc 所有用户应用程序都将链接到的 C 库。避免使用任何 C 库函数的内核和其它应用程序可以在没有该库的情况下进行编译。

构建工具链建立了一个交叉编译器环境。**本地编译器**编译与本机同类的处理器的指令。**交叉编译器**运行在某一种处理器上，却可以编译另一种处理器的指令。重头设置交叉编译器工具链可不是一项简单的任务：它包括下载源代码、修补补丁、配置、编译、设置头文件、安装以及很多很多的操作。另外，这样一个彻底的构建过程对内存和硬盘的需求是巨大的。如果没有足够的内存和硬盘空间，那么在构建阶段由于相关性、配置或头文件设置等问题会突然冒出许多问题。

因此能够从因特网上获得已预编译的二进制文件是一件好事（但不太好的一点是，目前它们大多数只限于基于 ARM 的系统，但迟早会改变的）。一些比较流行的已预编译的工具链包括那些来自 Compaq（Familiar Linux ）、LART（LART Linux）和 Embedian（基于 Debian 但与它无关）的工具链 — 所有这些工具链都用于基于 ARM 的平台。

内核设置

Linux 社区正积极地为新硬件添加功能部件和支持、在内核中修正错误并且及时地进行常规改进。这导致大约每 6 个月（或 6 个月不到）就有一个稳定的 Linux 树的新发行版。

不同的维护者维护针对特定体系结构的不同内核树和补丁。当为一个项目选择了一个内核时，您需要评估最新发行版的稳定性如何、它是否符合项目要求和硬件平台、从编程角度来看它的舒适程度以及其它难以确定的方面。还有一点也非常重要：找到需要应用于基本内核的所有补丁，以便为特定的体系结构调整内核。

内核布局

内核布局分为特定于体系结构的部分和与体系结构无关的部分。内核中特定于体系结构的部分首先执行，设置硬件寄存器、配置内存映像、执行特定于体系结构的初始化，然后将控制转给内核中与体系结构无关的部分。系统的其余部分在这第二个阶段期间进行初始化。内核树下的目录 arch/ 由不同的子目录组成，每个子目录用于一个不同的体系结构（MIPS、ARM、i386、SPARC、PPC 等）。每一个这样的子目录都包含 kernel/ 和 mm/ 子目录，它们包含特定于体系结构的代码来完成象初始化内存、设置 IRQ、启用高速缓存、设置内核页面表等操作。一旦装入内核并给予其控制，就首先调用这些函数，然后初始化系统的其余部分。

根据可用的系统资源和引导装载程序的功能，内核可以编译成 vmlinux、Image 或 zImage。vmlinux 和 zImage 之间的主要区别在于 *vmlinux* 是实际的（未压缩的）可执行文件，而 *zImage* 是或多或少包含相同信息的自解压压缩文件 — 只是压缩它以处理（通常是 Intel 强制的）640 KB 引导时间的限制。有关所有这些的权威性解释，请参阅 *Linux Magazine* 的文章“Kernel Configuration: dealing with the unexpected”（请参阅参考资料）。

内核链接和装入

一旦为目标系统编译了内核后，通过使用引导装载程序（它已经被装入到目标的闪存中），内核就被装入到目标系统的内存（在 DRAM 中或者在闪存中）。通过使用串行、USB 或以太网端口，引导装载程序与主机通信以将内核传送到目标的闪存或 DRAM 中。在将内核完全装入目标后，引导装载程序将控制传递给装入内核的地址。

内核可执行文件由许多链接在一起的对象文件组成。对象文件有许多节，如文本、资料、init 资料、bass 等等。这些对象文件都是由一个称为**链接器脚本**的文件链接并装入的。这个链接器脚本的功能是将输入对象文件的各节映像到输出文件中；换句话说，它将所有输入对象文件都链接到单一的可执行文件中，将该可执行文件的各节装入到指定地址处。
vmlinux.lds 是存在于 arch/<target>/ 目录中的内核链接器脚本，它负责链接内核的各个节并将它们装入内存中特定偏移量处。典型的 vmlinux.lds 看起来象这样：

清单 2. 典型的 vmlinux.lds 文件

```
OUTPUT_ARCH(<arch>)      /* <arch> includes architecture type */
 ENTRY(stext)               /* stext is the kernel entry point */
 SECTIONS                   /* SECTIONS command describes the layout
                  of the output file */
 {
     .  = TEXTADDR;         /* TEXTADDR is LMA for the kernel */
     .init : {          /* Init code and data*/
             _stext = .;       /* First section is stext followed
                  by __init data section */
             __init_begin = .;
```

```
                    *(.text.init)
             __init_end = .;
            }
     .text : {          /* Real text segment follows __init_data section */
             _text = .;
                    *(.text)
             _etext = .;       /* End of text section*/
            }
     .data :{
             _data=.;          /* Data section comes after text section */
                    *(.data)
             _edata=.;
            }                  /* Data section ends here */
     .bss : {                  /* BSS section follows symbol table section */
             __bss_start = .;
                    *(.bss)
             _end = . ;        /* BSS section ends here */
            }
  }
```

LMA 是装入模块地址；它表示将要装入内核的目标虚拟内存中的地址。TEXTADDR 是内核的虚拟起始地址，并且在 arch/<target>/ 下的 Makefile 中指定它的值。这个地址必须与引导装载程序使用的地址相匹配。

一旦引导装载程序将内核复制到闪存或 DRAM 中，内核就被重新定位到 TEXTADDR —它通常在 DRAM 中。然后，引导装载程序将控制转给这个位址，以便内核能开始执行。

参数传递和内核引导

stext 是内核入口点，这意味着在内核引导时将首先执行这一节下的代码。它通常用汇编语言编写，并且通常它在 arch/<target>/ 内核目录下。这个代码设置内核页面目录、创建身份内核映像、标识体系结构和处理器以及执行分支 start_kernel（初始化系统的主例程）。

start_kernel 调用 setup_arch 作为执行的第一步，在其中完成特定于体系结构的设置。这包括初始化硬件寄存器、标识根设备和系统中可用的 DRAM 和闪存的数量、指定系统中可用页面的数目、文件系统大小等等。所有这些信息都以参数形式从引导装载程序传递到内核。

将参数从引导装载程序传递到内核有两种方法：parameter_structure 和标记列表。在这两种方法中，不赞成使用参数结构，因为它强加了限制：指定在内存中，每个参数必须位于 param_struct 中的特定偏移量处。最新的内核期望参数作为标记列表的格式来传递，并将参数转化为已标记格式。param_struct 定义在 include/asm/setup.h 中。它的一些重要字段是：

清单 3. 样本参数结构

```
struct param_struct  {
```

```
 unsigned long page_size;     /* 0:  Size of the page  */
 unsigned long nr_pages;      /* 4:  Number of pages in the system */
 unsigned long ramdisk        /* 8: ramdisk size */
 unsigned long rootdev;       /* 16: Number representing the root device */
 unsigned long initrd_start;  /* 64: starting address of initial ramdisk */
                                     /* This can be either in flash/dram */
 unsigned long initrd_size;   /* 68: size of initial ramdisk */
}
```

请注意：这些数表示定义字段的参数结构中的偏移量。这意味着如果引导装载程序将参数结构放置在地址 0xc0000100，那么 rootdev 参数将放置在 0xc0000100 + 16，initrd_start 将放置在 0xc0000100 + 64 等等 — 否则，内核将在解释正确的参数时遇到困难。

正如上面提到的，因为从引导装载程序到内核的参数传递会有一些约束条件，所以大多数 2.4.x 系列内核期望参数以已标记的列表格式传递。在已标记的列表中，每个标记由标识被传递参数的 tag_header 以及其后的参数值组成。标记列表中标记的常规格式可以如下所示：

清单 4. 样本标记格式。内核通过 <ATAG_TAGNAME> 头来标识每个标记。

```
#define <aTAG_TAGNAME>  <Some Magic number>

struct <tag_tagname> {
        u32 <tag_param>;
        u32 <tag_param>;
};

/* Example tag for passing memory information */

#define ATAG_MEM        0x54410002  /* Magic number */

struct tag_mem32 {
        u32     size;               /* size of memory */
        u32     start;              /* physical start address of memory*/
};
```

setup_arch 还需要对闪存存储库、系统寄存器和其它特定设备执行内存映像。一旦完成了特定于体系结构的设置，控制就返回到初始化系统其余部分的 start_kernel 函数。这些附加的初始化任务包含：

- 设置陷阱
- 初始化中断
- 初始化定时器
- 初始化控制台

- 调用 mem_init，它计算各种区域、高内存区等内的页面数量
- 初始化 slab 分配器并为 VFS、缓冲区高速缓存等创建 slab 高速缓存
- 建立各种文件系统，如 proc、ext2 和 JFFS2
- 创建 kernel_thread，它执行文件系统中的 init 命令并显示 lign 提示符。 如果在 /bin、/sbin 或 /etc 中没有 init 程序，那么内核将执行文件系统的 /bin 中的 shell。

三、设置驱动程序

嵌入式系统通常有许多设备用于与用户交互，象触摸屏、小键盘、滚动轮、传感器、RA232 接口、LCD 等等。除了这些设备外，还有许多其它专用设备，包括闪存、USB、GSM 等。内核通过所有这些设备各自的设备驱动程序来控制它们，包括 GUI 用户应用程序也通过访问这些驱动程序来访问设备。本节着重讨论通常几乎在每个嵌入式环境中都会使用的一些重要设备的设备驱动程序。

帧缓冲区驱动程序

这是最重要的驱动程序之一，因为通过这个驱动程序才能使系统屏幕显示内容。帧缓冲区驱动程序通常有三层。最底层是基本控制台驱动程序 drivers/char/console.c，它提供了文本控制台常规接口的一部分。通过使用控制台驱动程序函数，我们能将文本打印到屏幕上 — 但图形或动画还不能（这样做需要使用视频模式功能，通常出现在中间层，也就是 drivers/video/fbcon.c 中）。这个第二层驱动程序提供了视频模式中绘图的常规接口。帧缓冲区是显卡上的内存，需要将它内存映像到用户空间以便可以将图形和文本能写到这个内存段上：然后这个信息将反映到屏幕上。帧缓冲区支持提高了绘图的速度和整体性能。这也是顶层驱动程序引人注意之处：顶层是非常特定于硬件的驱动程序，它需要支持显卡不同的硬件方面 — 象启用 / 禁用显卡控制器、深度和模式的支持以及调色板等。所有这三层都相互依赖以实现正确的视频功能。与帧缓冲区有关的设备是 /dev/fb0（主设备号 29，次设备号 0）。

输入设备驱动程序

可触摸板是用于嵌入式设备的最基本的用户交互设备之一 — 小键盘、传感器和滚动轮也包含在许多不同设备中以用于不同的用途。

触摸板设备的主要功能是随时报告用户的触摸，并标识触摸的坐标。这通常在每次发生触摸时，通过生成一个中断来实现。

然后，这个设备驱动程序的角色是每当出现中断时就查询触摸屏控制器，并请求控制器发送触摸的坐标。一旦驱动程序接收到坐标，它就将有关触摸和任何可用资料的信号发送给用户应用程序，并将资料发送给应用程序（如果可能的话）。然后用户应用程序根据它的需要处理资料。

几乎所有输入设备 — 包括小键盘 — 都以类似原理工作。

闪存 MTD 驱动程序

MTD 设备是象闪存芯片、小型闪存卡、记忆棒等之类的设备，它们在嵌入式设备中的使用正在不断增长。

MTD 驱动程序是在 Linux 下专门为嵌入式环境开发的新的一类驱动程序。相对于常规块设备驱动程序，使用 MTD 驱动程序的主要优点在于 MTD 驱动程序是专门为基于闪存的设备所设计的，所以它们通常有更好的支持、更好的管理和基于扇区的擦除和读写操作的更好的接口。Linux 下的 MTD 驱动程序接口被划分为两类模块：用户模块和硬件模块。

用户模块

这些模块提供从用户空间直接使用的接口：原始字符访问、原始块访问、FTL（闪存转换层，Flash Transition Layer — 用在闪存上的一种文件系统）和 JFS（即日志文件系统，Journaled File System — 在闪存上直接提供文件系统而不是模拟块设备）。用于闪存的 JFS 的当前版本是 JFFS2（稍后将在本文中描述）。

硬件模块

这些模块提供对内存设备的物理访问，但并不直接使用它们。通过上述的用户模块来访问它们。这些模块提供了在闪存上读、擦除和写操作的实际例程。

MTD 驱动程序设置

为了访问特定的闪存设备并将文件系统置于其上，需要将 MTD 子系统编译到内核中。这包括选择适当的 MTD 硬件和用户模块。当前,MTD 子系统支持为数众多的闪存设备 — 并且有越来越多的驱动程序正被添加进来以用于不同的闪存芯片。

有两个流行的用户模块可启用对闪存的访问：MTD_CHAR 和 MTD_BLOCK。

MTD_CHAR 提供对闪存的原始字符访问，而 MTD_BLOCK 将闪存设计为可以在上面创建文件系统的常规块设备（象 IDE 磁盘）。与 MTD_CHAR 关联的设备是 /dev/mtd0、mtd1、mtd2（等等），而与 MTD_BLOCK 关联的设备是 /dev/mtdblock0、mtdblock1（等等）。由于 MTD_BLOCK 设备提供象块设备那样的模拟，通常更可取的是在这个模拟基础上创建象 FTL 和 JFFS2 那样的文件系统。

为了进行这个操作，可能需要创建分区表将闪存设备分拆到引导装载程序节、内核节和文件系统节中。样本分区表可能包含以下信息：

清单 5. MTD 的简单闪存设备分区

```
struct mtd_partition sample_partition = {
      {
                                    /* First partition */
            name : bootloader,            /* Bootloader section */
            size    : 0x00010000,         /* Size  */
            offset  : 0,          /* Offset from start of flash- location 0x0*/
            mask_flags : MTD_WRITEABLE     /* This partition is not writable */
      },
      {                             /* Second partition */
            name : Kernel,                /* Kernel section */
            size    :  0x00100000,        /* Size */
            offset : MTDPART_OFS_APPEND,   /* Append after bootloader section */
            mask_flags : MTD_WRITEABLE     /* This partition is not writable */
      },
      {                             /* Third partition */
            name : JFFS2,                 /* JFFS2 filesystem */
            size    :  MTDPART_SIZ_FULL,  /* Occupy rest of flash */
            offset :  MTDPART_OFS_APPEND   /* Append after kernel section */
      }
 }
```

上面的分区表使用了 MTD_BLOCK 接口对闪存设备进行分区。这些分区的设备节点是：

简单闪存分区的设备节点

```
User      device node        Major number    Minor number

 Bootloader    /dev/mtdblock0         31              0
 Kernel        /dev/mtdblock1         31              1
 Filesystem    /dev/mtdblock2         31              2
```

在本例中，引导装载程序必须将有关 root 设备节点（/dev/mtdblock2）和可以在闪存中找到文件系统的地址（本例中是 FLASH_BASE_ADDRESS + 0x04000000）的正确参数传递到内核。一旦完成分区，闪存设备就准备装入或挂装文件系统。

Linux 中 MTD 子系统的主要目标是在系统的硬件驱动程序和上层，或用户模块之间提供通用接口。硬件驱动程序不需要知道象 JFFS2 和 FTL 那样的用户模块使用的方法。所有它们真正需要提供的就是一组对底层闪存系统进行 read、 write 和 erase 操作的简单例程。

四、嵌入式设备的文件系统

系统需要一种以结构化格式存储和检索信息的方法；这就需要文件系统的参与。Ramdisk（请参阅参考资料）是通过将计算机的 RAM 用作设备来创建和挂装文件系统的一种机制，它通常用于无盘系统（当然包括微型嵌入式设备，它只包含作为永久存储媒质的闪存芯片）。

用户可以根据可靠性、健壮性和 / 或增强的功能的需求来选择文件系统的类型。下一节将讨论几个可用选项及其优缺点。

第二版扩展文件系统（Ext2fs）

xt2fs 是 Linux 事实上的标准文件系统，它已经取代了它的前任 — 扩展文件系统（或 Extfs）。Extfs 支持的文件大小最大为 2 GB，支持的最大文件名称大小为 255 个字符 — 而且它不支持索引节点（包括资料修改时间标记）。Ext2fs 做得更好；它的优点是：

- Ext2fs 支持达 4 TB 的内存。
- Ext2fs 文件名称最长可以到 1012 个字符。
- 当创建文件系统时，管理员可以选择逻辑块的大小（通常大小可选择 1024、2048 和 4096 字节）。
- Ext2fs 了实现快速符号链接：不需要为此目的而分配数据块，并且将目标名称直接存储在索引节点（inode）表中。这使性能有所提高，特别是在速度上。

因为 Ext2 文件系统的稳定性、可靠性和健壮性，所以几乎在所有基于 Linux 的系统（包括台式机、服务器和工作站 — 并且甚至一些嵌入式设备）上都使用 Ext2 文件系统。然而，当在嵌入式设备中使用 Ext2fs 时，它有一些缺点：

- Ext2fs 是为象 IDE 设备那样的块设备设计的，这些设备的逻辑块大小是 512 字节，1 K 字节等这样的倍数。这不太适合于扇区大小因设备不同而不同的闪存设备。
- Ext2 文件系统没有提供对基于扇区的擦除 / 写操作的良好管理。在 Ext2fs 中，为了在一个扇区中擦除单个字节，必须将整个扇区复制到 RAM，然后擦除，然后重写入。考虑到闪存设备具有有限的擦除寿命（大约能进行 100,000 次擦除），在此

之后就不能使用它们，所以这不是一个特别好的方法。

- 在出现电源故障时，Ext2fs 不是防崩溃的。
- Ext2 文件系统不支持损耗平衡，因此缩短了扇区 / 闪存的寿命。（损耗平衡确保将地址范围的不同区域轮流用于写和 / 或擦除操作以延长闪存设备的寿命。）
- Ext2fs 没有特别完美的扇区管理，这使设计块驱动程序十分困难。

由于这些原因，通常相对于 Ext2fs，在嵌入式环境中使用 MTD/JFFS2 组合是更好的选择。

用 Ramdisk 挂装 Ext2fs

通过使用 Ramdisk 的概念，可以在嵌入式设备中创建并挂装 Ext2 文件系统（以及用于这一目的的任何文件系统）。

清单 6. 创建一个简单的基于 Ext2fs 的 Ramdisk

```
mke2fs -vm0 /dev/ram 4096
 mount -t ext2 /dev/ram /mnt
 cd /mnt
 cp /bin, /sbin, /etc, /dev ... files in mnt
 cd ../
 umount /mnt
 dd if=/dev/ram bs=1k count=4096 of=ext2ramdisk
```

mke2fs 是用于在任何设备上创建 ext2 文件系统的实用程序 — 它创建超级块、索引节点以及索引节点表等等。

在上面的用法中，/dev/ram 是上面构建有 4096 个块的 ext2 文件系统的设备。然后，将这个设备（/dev/ram）挂装在名为 /mnt 的临时目录上并且复制所有必需的文件。一旦复制完这些文件，就卸装这个文件系统并且设备（/dev/ram）的内容被转储到一个文件（ext2ramdisk）中，它就是所需的 Ramdisk（Ext2 文件系统）。

上面的顺序创建了一个 4 MB 的 Ramdisk，并用必需的文件实用程序来填充它。

一些要包含在 Ramdisk 中的重要目录是：

- **/bin** — 保存大多数象 init、busybox、shell、文件管理实用程序等二进制文件。
- **/dev** — 包含用在设备中的所有设备节点
- **/etc** — 包含系统的所有配置文件
- **/lib** — 包含所有必需的库，如 libc、libdl 等

日志闪存文件系统，版本 2（JFFS2）

瑞典的 Axis Communications 开发了最初的 JFFS，Red Hat 的 David Woodhouse 对它进行了改进。 第二个版本，JFFS2，作为用于微型嵌入式设备的原始闪存芯片的实际文件系统而出现。JFFS2 文件系统是日志结构化的，这意味着它基本上是一长列节点。每个节点包含有关文件的部分信息 — 可能是文件的名称、也许是一些资料。相对于 Ext2fs，JFFS2 因为有以下这些优点而在无盘嵌入式设备中越来越受欢迎：

- JFFS2 在扇区级别上执行闪存擦除 / 写 / 读操作要比 Ext2 文件系统好。
- JFFS2 提供了比 Ext2fs 更好的崩溃 / 掉电安全保护。当需要更改少量资料时，Ext2 文件系统将整个扇区复制到内存（DRAM）中，在内存中合并新资料，并写回整个扇区。这意味着为了更改单个字，必须对整个扇区（64 KB）执行读 / 擦除 / 写例程 — 这样做的效率非常低。要是运气差，当正在 DRAM 中合并资料时，发生了电源故障或其它事故，那么将丢失整个资料集合，因为在将资料读入 DRAM 后就擦除了闪存扇区。JFFS2 附加文件而不是重写整个扇区，并且具有崩溃 / 掉电安全保护这一功能。
- 这可能是最重要的一点：JFFS2 是专门为象闪存芯片那样的嵌入式设备创建的，所以它的整个设计提供了更好的闪存管理。

因为本文主要是写关于闪存设备的使用，所以在嵌入式环境中使用 JFFS2 的缺点很少：

- 当文件系统已满或接近满时，JFFS2 会大大放慢运行速度。这是因为垃圾收集的问题（更多信息，请参阅参考资料）。

创建 JFFS2 文件系统

在 Linux 下，用 mkfs.jffs2 命令创建 JFFS2 文件系统（基本上是使用 JFFS2 的 Ramdisk）。

清单 7. 创建 JFFS2 文件系统

```
mkdir jffsfile
 cd jffsfile

 /* copy all the /bin, /etc, /usr/bin, /sbin/ binaries and /dev entries
that are needed for the filesystem here */

 /* Type the following command under jffsfile directory to create the JFFS2 Image */

 ./mkfs.jffs2 -e 0x40000 -p -o ../jffs.image
```

上面显示了 mkfs.jffs2 的典型用法。-e 选项确定闪存的擦除扇区大小（通常是 64 千字节）。-p 选项用来在映像的剩余空间用零填充。-o 选项用于输出文件，通常是 JFFS2 文件系统映像 — 在本例中是 jffs.image。一旦创建了 JFFS2 文件系统，它就被装入闪存中适当的位置（引导装载程序告知内核查找文件系统的地址）以便内核能挂装它。

tmpfs

当 Linux 运行于嵌入式设备上时，该设备就成为功能齐全的单元，许多守护进程会在后台运行并生成许多日志消息。另外，所有内核日志记录机制，象 syslogd、dmesg 和 klogd，会在 /var 和 /tmp 目录下生成许多消息。由于这些进程产生了大量资料，所以允许将所有这些写操作都发生在闪存是不可取的。由于在重新引导时这些消息不需要持久存储，所以这

个问题的解决方案是使用 tmpfs。

tmpfs 是基于内存的文件系统,它主要用于减少对系统的不必要的闪存写操作这一唯一目的。因为 tmpfs 驻留在 RAM 中，所以写 / 读 / 擦除的操作发生在 RAM 中而不是在闪存中。因此，日志消息写入 RAM 而不是闪存中，在重新引导时不会保留它们。tmpfs 还使用磁盘交换空间来存储，并且当为存储文件而请求页面时，使用虚拟内存（VM）子系统。

tmpfs 的**优点**包括：

- 动态文件系统大小 — 文件系统大小可以根据被复制、创建或删除的文件或目录的数量来缩放。使得能够最理想地使用内存。
- 速度 — 因为 tmpfs 驻留在 RAM，所以读和写几乎都是瞬时的。即使以交换的形式存储文件，I/O 操作的速度仍非常快。

tmpfs 的一个**缺点**是当系统重新引导时会丢失所有资料。因此，重要的资料不能存储在 tmpfs 上。

挂装 tmpfs

诸如 Ext2fs 和 JFFS2 等大多数其它文件系统都驻留在底层块设备之上,而 tmpfs 与它们不同，它直接位于 VM 上。因而，挂装 tmpfs 文件系统是很简单的事：

清单 8. 挂装 tmpfs

```
/* Entries in /etc/rc.d/rc.sysinit for creating/using tmpfs */


 # mount -t tmpfs tmpfs /var -o size=512k
 # mkdir -p /var/tmp
 # mkdir -p /var/log
 # ln -s /var/tmp /tmp
```

上面的命令将在 /var 上创建 tmpfs 并将 tmpfs 的最大大小限制为 512 K。同时，tmp/ 和 log/ 目录成为 tmpfs 的一部分以便在 RAM 中存储日志消息。

如果您想将 tmpfs 的一个项添加到 /etc/fstab，那么它可能看起来象这样：

tmpfs /var tmpfs size=32m 0 0

这将在 /var 上挂装一个新的 tmpfs 文件系统。

五、图形用户接口(GUI)选项

从用户的观点来看,图形用户接口(GUI)是系统的一个最至关重要的方面:用户通过 GUI 与系统进行交互。所以 GUI 应该易于使用并且非常可靠。但它还需要是有内存意识的，以便在内存受限的、微型嵌入式设备上可以无缝执行。所以，它应该是轻量级的，并且能够快速装入。

另一个要考虑的重要方面涉及许可证问题。一些 GUI 分发版具有允许免费使用的许可证，甚至在一些商业产品中也是如此。另一些许可证要求如果想将 GUI 合并入项目中则要支付版税。

最后，大多数开发人员可能会选择 XFree86，因为 XFree86 为他们提供了一个能使用他们喜欢的工具的熟悉环境。但是市场上较新的 GUI，象 Century Software 的 Microwindows（Nano-X）和 Trolltech 的 QT/Embedded，与 X 在嵌入式 Linux 的竞技舞台中展开了激烈竞争，这主要是因为它们占用很少的资源、执行的速度很快并且具有定制窗口构件的支持。

让我们看一看这些选项中的每一个。

Xfree86 4.X（带帧缓冲区支持的 X11R6.4）

XFree86 Project, Inc. 是一家生产 XFree86 的公司，该产品是一个可以免费重复分发、开放源码的 X Window 系统。X Window 系统（X11）为应用程序以图形方式进行显示提供了资源，并且它是 UNIX 和类 UNIX 的机器上最常用的窗口系统。它很小但很有效，它运行在为数众多的硬件上，它对网络透明并且有良好的文档说明。X11 为窗口管理、事件处理、同步和客户机间通信提供强大的功能 — 并且大多数开发人员已经熟悉了它的 API。它具有对内核帧缓冲区的内置支持，并占用非常少的资源 — 这非常有助于内存相对较少的设备。X 服务器支持 VGA 和非 VGA 图形卡，它对颜色深度 1、2、4、8、16 和 32 提供支持，并对渲染提供内置支持。最新的发行版是 XFree86 4.1.0。

它的优点包括：

- 帧缓冲区体系结构的使用提高了性能。
- 占用的资源相对很小 — 大小在 600 K 到 700 K 字节的范围内，这使它很容易在小型设备上运行。
- 非常好的支持：在线有许多文档可用，还有许多专用于 XFree86 开发的邮递列表。
- X API 非常适合扩展。

它的缺点包括：

- 比最近出现的嵌入式 GUI 工具性能差。
- 此外，当与 GUI 中最新的开发 — 象专门为嵌入式环境设计的 Nano-X 或 QT/Embedded — 相比时，XFree86 似乎需要更多的内存。

Microwindows

Microwindows 是 Century Software 的开放源代码项目，设计用于带小型显示单元的微型设备。它有许多针对现代图形窗口环境的功能部件。象 X 一样，有多种平台支持 Microwindows。

Microwindows 体系结构是基于客户机／服务器的并且具有分层设计。最底层是屏幕和输入设备驱动程序（关于键盘或鼠标）来与实际硬件交互。在中间层，可移植的图形引擎提供对线的绘制、区域的填充、多边形、裁剪以及颜色模型的支持。

在最上层，Microwindows 支持两种 API：Win32/WinCE API 实现，称为 Microwindows；另一种 API 与 GDK 非常相似，它称为 Nano-X。Nano-X 用在 Linux 上。它是象 X 的 API，用于占用资源少的应用程序。

Microwindows 支持 1、2、4 和 8 bpp（每像素的位数）的 palletized 显示，以及 8、16、24 和 32 bpp 的真彩色显示。Microwindows 还支持使它速度更快的帧缓冲区。Nano-X 服务器占用的资源大约在 100 K 到 150 K 字节。

原始 Nano-X 应用程序的平均大小在 30 K 到 60 K。由于 Nano-X 是为有内存限制的低端设备设计的,所以它不象 X 那样支持很多函数,因此它实际上不能作为微型 X(Xfree86 4.1）的替代品。

可以在 Microwindows 上运行 FLNX,它是针对 Nano-X 而不是 X 进行修改的 FLTK(快速轻巧工具箱(Fast Light Toolkit)）应用程序开发环境的一个版本。本文中描述 FLTK。

Nano-X 的优点包括：

- 与 Xlib 实现不同，Nano-X 仍在每个客户机上同步运行，这意味着一旦发送了客户机请求包，服务器在为另一个客户机提供服务之前一直等待，直到整个包都到达为止。这使服务器代码非常简单，而运行的速度仍非常快。
- 占用很小的资源

Nano-X 的缺点包括：
联网功能部件至今没有经过适当地调整（特别是网络透明性）。

- 还没有太多现成的应用程序可用。
- 与 X 相比，Nano-X 虽然近来正在加速开发，但仍没有那么多文档说明而且没有很好的支持，但这种情形会有所改变。

Microwindows 上的 FLTK API

FLTK 是一个简单但灵活的 GUI 工具箱，它在 Linux 世界中赢得越来越多的关注，它特别适用于占用资源很少的环境。它提供了您期望从 GUI 工具箱中获得的大多数窗口构件，如按钮、对话框、文本框以及出色的“赋值器”选择（用于输入数值的窗口构件）。还包括滑动器、滚动条、刻度盘和其它一些构件。

针对 Microwindows GUI 引擎的 FLTK 的 Linux 版本被称为 FLNX。FLNX 由两个组件构成：Fl_Widget 和 FLUID。Fl_Widget 由所有基本窗口构件 API 组成。FLUID（快速轻巧的用户接口设计器(Fast Light User Interface Designer, FLUID)）是用来产生 FLTK 源代码的图形编辑器。总的来说，FLNX 是能用来为嵌入式环境创建应用程序的一个出色的 UI 构建器。

Fl_Widget 占用的资源大约是 40 K 到 48 K，而 FLUID（包括了每个窗口构件）大约占用 380 K。这些非常小的资源占用率使 Fl_Widget 和 FLUID 在嵌入式开发世界中非常受欢迎。
优点包括：

- 习惯于在象 Windows 这样已建立得较好的环境中开发基于 GUI 的应用程序的任何人都会非常容易地适应 FLTK 环境。
- 它的文檔包括一本十分完整且编写良好的手册。
- 它使用 LGPL 进行分发，所以开发人员可以灵活地发放他们应用程序的许可证。
- FLTK 是一个 C++ 库（Perl 和 Python 绑定也可用）。面向对象模型的选择是一个

好的选择，因为大多数现代 GUI 环境都是面向对象的；这也使将编写的应用程序移植到类似的 API 中变得更容易。

- Century Software 的环境提供了几个有用的工具，诸如 ScreenToP 和 ViewML 浏览器。

它的缺点是：

- 普通的 FLTK 可以与 X 和 Windows API 一同工作，而 FLNX 不能。它与 X 的不兼容性阻碍了它在许多项目中的使用。

Qt/Embeded

Qt/Embedded 是 Trolltech 新开发的用于嵌入式 Linux 的图形用户接口系统。Trolltech 最初创建 Qt 作为跨平台的开发工具用于 Linux 台式机。它支持各种有 UNIX 特点的系统以及 Microsoft Windows。KDE — 最流行的 Linux 桌面环境之一，就是用 Qt 编写的。

Qt/Embedded 以原始 Qt 为基础，并做了许多出色的调整以适用于嵌入式环境。Qt Embedded 通过 Qt API 与 Linux I/O 设施直接交互。那些熟悉并已适应了面向对象编程的人员将发现它是一个理想环境。而且，面向对象的体系结构使代码结构化、可重用并且运行快速。与其它 GUI 相比，Qt GUI 非常快，并且它没有分层，这使得 Qt/Embedded 成为用于运行基于 Qt 的程序的最紧凑环境。

Trolltech 还推出了 Qt 掌上机环境（Qt Palmtop Environment，俗称 Qpe）。Qpe 提供了一个基本桌面窗口，并且该环境为开发提供了一个易于使用的接口。Qpe 包含全套的个人信息管理（Personal Information Management (PIM)）应用程序、因特网客户机、实用程序等等。然而，为了将 Qt/Embedded 或 Qpe 集成到一个产品中，需要从 Trolltech 获得商业许可证。（原始 Qt 自版本 2.2 以后就可以根据 GPL 获得 。）

它的优点包括：

- 面向对象的体系结构有助于更快地执行
- 占用很少的资源，大约 800 K
- 抗锯齿文本和混合视频的象素映像

它的缺点是：

- Qt/Embedded 和 Qpe 只能在获得商业许可证的情况下才能使用。

第十六章 LINUX 下 BOOTLOADER 的开发

一个嵌入式 Linux 系统从软件的角度看通常可以分为四个层次：

1. **引导加载程序**。包括固化在固件(firmware)中的 boot 代码(可选)，和 Boot Loader 两大部分。
2. **Linux 内核**。特定于嵌入式板子的定制内核以及内核的启动参数。
3. **文件系统**。包括根文件系统和建立于 Flash 内存设备之上文件系统。我们用 cramfs 来

作为 root fs。

4. **用户应用程序。**特定于用户的应用程序。有时在用户应用程序和内核层之间可能还会包括一个嵌入式图形用户接口。我们使用的是 QT.

在一个基于 ARM7TDMI core 的嵌入式系统中，系统在上电或复位时通常都从地址 0x00000000 处开始执行，而在这个地址处安排的通常就是系统的 Boot Loader 程序。Boot Loader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬设备、建立内存空间的映像图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。

一、Boot Loader 概念

通常 Boot Loader 是严重地依赖于硬件而实现的，特别是在嵌入式世界。因此，在嵌入式世界里建立一个通用的 Boot Loader 几乎是不可能的。尽管如此，我们仍然可以对 Boot Loader 归纳出一些通用的概念来，以指导用户特定的 Boot Loader 设计与实现。

二、Boot Loader 所支持的 cpu 和嵌入板

每种不同的 CPU 体系结构都有不同的 Boot Loader。有些 Boot Loader 也支持多种体系结构的 CPU，比如 U-Boot 就同时支持 ARM 体系结构和 MIPS 体系结构。除了依赖于 CPU 的体系结构外，Boot Loader 实际上也依赖于具体的嵌入式板级设备的配置。这也就是说，对于两块不同的嵌入式板而言，即使它们是基于同一种 CPU 而构建的，要想让运行在一块板子上的 Boot Loader 程序也能运行在另一块板子上，通常也都需要修改 Boot Loader 的源程序。

三、Boot Loader 的安装媒介（Installation Medium）

系统加电或复位后，所有的 CPU 通常都从某个由 CPU 制造商预先安排的位址上取指令。比如，基于 ARM core 的 CPU 在复位时通常都从地址 0x00000000 取它的第一条指令。而基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储设备(比如：ROM、EEPROM 或 FLASH 等)被映像到这个预先安排的地址上。因此在系统加电后，CPU 将首先执行 Boot Loader 程序。

下图 1 就是一个同时装有 Boot Loader、内核的启动参数、内核映像和根文件系统映像的固态存储设备的典型空间分配结构图。

图 1 固态存储设备的典型空间分配结构

四、Boot Loader 来控制 Boot Loader 的设备或机制

主机和目标机之间一般通过串口建立连接，Boot Loader 软件在执行时通常会通过串口来进行 I/O，比如：输出打印信息到串口，从串口读取用户控制字符等。

五、Boot Loader 的启动过程是单阶段（Single Stage）还是多阶段（Multi-Stage）。

通常多阶段的 Boot Loader 能提供更为复杂的功能，以及更好的可移植性。从固态存储设备上启动的 Boot Loader 大多都是 2 阶段的启动过程，也即启动过程可以分为 stage 1 和 stage 2 两部分。而至于在 stage 1 和 stage 2 具体完成哪些任务将在下面几篇讨论。

六、Boot Loader 的操作模式（Operation Mode）大多数

Boot Loader 都包含两种不同的操作模式："启动加载"模式和"下载"模式，这种区别仅对于开发人员才有意义。但从最终用户的角度看，Boot Loader 的作用就是用来加载操作系统，而并不存在所谓的启动加载模式与下载工作模式的区别。

启动加载（Boot loading）模式：这种模式也称为"自主"（Autonomous）模式。也即 Boot Loader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行，整个过程并没有用户的介入。这种模式是 Boot Loader 的正常工作模式，因此在嵌入式产品发布的时侯，Boot Loader 显然必须工作在这种模式下。

下载（Downloading）模式：在这种模式下，目标机上的 Boot Loader 将通过串口连接或网络连接等通信手段从主机（Host）下载文件，比如：下载内核映像和根文件系统映像等。从主机下载的文件通常首先被 Boot Loader 保存到目标机的 RAM 中，然后再被 Boot Loader 写到目标机上的 FLASH 类固态存储设备中。Boot Loader 的这种模式通常在第一次安装内核与根文件系统时被使用；此外，以后的系统更新也会使用 Boot Loader 的这种工作模式。工作于这种模式下的 Boot Loader 通常都会向它的终端用户提供一个简单的命令行接口。 我们编写的 BootLoader vivi 只支持启动加载（Boot loading）模式

七、Boot Loader 与主机进行文件传输所用的通信设备及协议

最常见的情况就是，目标机上的 Boot Loader 通过串口与主机之间进行文件传输，传输协议通常是 xmodem／ymodem／zmodem 协议中的一种。但是，串口传输的速度是有限的，因此通过以太网连接并借助 TFTP 协议来下载文件是个更好的选择。

八、Boot Loader vivi 的 stage1 通常包括以下步骤（以执行的先后顺序)：

• 硬设备初始化。

- 为加载 Boot Loader 的 stage2 准备 RAM 空间
- 拷贝 Boot Loader 的 stage2 到 RAM 空间中。
- 设置好堆栈。
- 跳转到 stage2 的 C 入口点。

九、Boot Loader vivi 的 stage2 通常包括以下步骤（以执行的先后顺序）：

- 初始化本阶段要使用到的硬设备。
- 检测系统内存映像(memory map)。
- 将 kernel 映像和根文件系统映像从 flash 上读到 RAM 空间中。
- 为内核设置启动参数。

第十七章 LINUX 驱动程序模块的编写

Linux 是 Unix 操作系统的一种变种，在 Linux 下编写驱动程序的原理和思想完全类似于其它的 Unix 系统，但它 dos 或 window 环境下的驱动程序有很大的区别。在 Linux 环境下设计驱动程序，思想简洁，操作方便，功能也很强大，但是支持函数少，只能依赖 kernel 中的函数，有些常用的操作要自己来编写，而且调试也不方便。

一、inux device driver 的概念

系统调用是操作系统内核和应用程序之间的接口,设备驱动程序是操作系统内核和机器硬件之间的接口.设备驱动程序为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬设备只是一个设备文件，应用程序可以象操作普通文件一样对硬设备进行操作.设备驱动程序是内核的一部分，它完成以下的功能:

1.对设备初始化和释放.

2.把资料从内核传送到硬件和从硬件读取资料.

3.读取应用程序传送给设备文件的资料和回送应用程序请求的资料.

4.检测和处理设备出现的错误.

在Linux操作系统下有两类主要的设备文件类型，一种是字符设备，另一种是块设备.字符设备和块设备的主要区别是:在对字符设备发出读/写请求时,实际的硬件I/O一般就紧接着发生了，块设备则不然，它利用一块系统内存作缓冲区，当用户进程对设备请求能满足用户的要求，就返回请求的资料，如果不能，就调用请求函数来进行实际的I/O操作.块设备是主要针对磁盘等慢速设备设计的，以免耗费过多的CPU时间来等待.

已经提到，用户进程是通过设备文件来与实际的硬件打交道.每个设备文件都都有其文件属性(c/b)，表示是字符设备还蔷强樯璞?另外每个文件都有两个设备号，第一个是主设备号，标识驱动程序，第二个是从设备号，标识使用同一个设备驱动程序的不同的硬设备，比如有两个软盘，就可以用从设备号来区分他们.设备文件的的主设备号必须与设备驱动程序在登记时申请的主设备号一致，否则用户进程将无法访问到驱动程序.

最后必须提到的是，在用户进程调用驱动程序时，系统进入核心态，这时不再是抢先式调度.也就是说,系统必须在你的驱动程序的子函数返回后才能进行其它的工作.如果你的驱动程序陷入死循环，不幸的是你只有重新启动机器了，然后就是漫长的fsck.

读/写时，它首先察看缓冲区的内容，如果缓冲区的资料

如何编写Linux操作系统下的设备驱动程序

二、实例剖析

由于用户进程是通过设备文件同硬件打交道,对设备文件的操作方式不外乎就是一些系统调用，如 open，read，write，close....， 注意，不是fopen， fread，但是如何把系统调用和驱动程序关联起来呢?这需要了解一个非常关键的数据结构:

```
struct file_operations {
int (*seek) (struct inode * , struct file *,  off_t , int);
int (*read) (struct inode * , struct file *,  char , int);
int (*write) (struct inode * , struct file *,  off_t , int);
int (*readdir) (struct inode * , struct file *,  struct dirent * , int);
int (*select) (struct inode * , struct file *,  int , select_table *);
int (*ioctl) (struct inode * , struct file *,  unsined int , unsigned long);
int (*mmap) (struct inode * , struct file *,  struct vm_area_struct *);
int (*open) (struct inode * , struct file *);
int (*release) (struct inode * , struct file *);
int (*fsync) (struct inode * , struct file *);
```

```
int (*fasync) (struct inode * , struct file *, int);
int (*check_media_change) (struct inode * , struct file *);
int (*revalidate) (dev_t dev);
}
```

这个结构的每一个成员的名字都对应着一个系统调用.用户进程利用系统调用在对设备文件进行诸如 read/write 操作时，系统调用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着把控制权交给该函数.这是 linux 的设备驱动程序工作的基本原理.既然是这样，则编写设备驱动程序的主要工作就是编写子函数，并填充 file_operations 的各个域.

三、设备驱动程序中的一些具体问题

I/O Port.

和硬件打交道离不开 I/O Port，老的 ISA 设备经常是占用实际的 I/O 埠，在 linux 下，操作系统没有对 I/O 口屏蔽，也就是说，任何驱动程序都可对任意的 I/O 口操作，这样就很容易引起混乱。每个驱动程序应该自己避免误用埠。

有两个重要的 kernel 函数可以保证驱动程序做到这一点。

1）check_region(int io_port， int off_set)

这个函数察看系统的 I/O 表，看是否有别的驱动程序占用某一段 I/O 口。

参数 1：io 端口的基地址，

参数 2：io 埠占用的范围。

返回值：0 没有占用， 非 0，已经被占用。

2）request_region(int io_port， int off_set，char *devname)

如果这段 I/O 埠没有被占用，在我们的驱动程序中就可以使用它。在使用之前，必须向系统登记，以防止被其它程序占用。登记后，在/proc/ioports 文件中可以看到你登记的 io 口。

参数 1：io 端口的基地址。

参数 2：io 埠占用的范围。

参数 3：使用这段 io 地址的设备名。

在对 I/O 口登记后，就可以放心地用 inb()， outb()之类的函来访问了。

在一些 pci 设备中，I/O 埠被映像到一段内存中去，要访问这些埠就相当于访问一段内存。经常性的，我们要获得一块内存的物理位址。在 dos 环境下，(之所以不说是 dos 操作系统是因为我认为 DOS 根本就不是一个操作系统，它实在是太简单，太不安全了)只要用段：偏移就可以了。在 window95 中，95ddk 提供了一个 vmm 调用 _MapLinearToPhys，用以把线性位址转化为物理位址。但在 Linux 中是怎样做的呢？

记忆体操作

在设备驱动程序中动态开辟内存，不是用 malloc，而是 kmalloc，或者用 get_free_pages 直接申请页。释放内存用的是 kfree，或 free_pages. 请注意，kmalloc 等函数返回的是物理位址！而 malloc 等返回的是线性地址！关于 kmalloc 返回的是物理位址这一点本人有点不太明白：既然从线性位址到物理位址的转换是由 386cpu 硬件完成的，那样汇编指令的操作数应该是线性地址，驱动程序同样也不能直接使用物理位址而是线性地址。但是事实上 kmalloc 返回的确实是物理位址，而且也可以直接通过它访问实际的 RAM，我想这样可以由两种解释，一种是在核心态禁止分页，但是这好象不太现实；另一种是 linux 的页目录和页表项设计得正好使得物理位址等同于线性地址。我的想法不知对不对，还请高手指教。

言归正传，要注意 kmalloc 最大只能开辟 128k-16，16 个字节是被页描述符结构占用了。

（kmalloc 用法参见 khg）

内存映像的 I/O 口，寄存器或者是硬设备的 RAM(如显存)一般占用 F0000000 以上的地址空间。在驱动程序中不能直接访问，要通过 kernel 函数 vremap 获得重新映像以后的地址。

另外，很多硬件需要一块比较大的连续内存用作 DMA 传送。这块内存需要一直驻留在内存，不能被交换到文件中去。但是 kmalloc 最多只能开辟 128k 的内存。

这可以通过牺牲一些系统内存的方法来解决。

具体做法是：比如说你的机器由 32M 的内存，在 lilo.conf 的启动参数中加上 mem=30M，这样 linux 就认为你的机器只有 30M 的内存，剩下的 2M 内存在 vremap 之后就可以为 DMA 所用了。

请记住，用 vremap 映像后的内存，不用时应用 unremap 释放，否则会浪费页表。

中断处理

同处理 I/O 埠一样，要使用一个中断，必须先向系统登记。

```
int request_irq(unsigned int irq ,
void(*handle)(int，void *，struct pt_regs *)，
unsigned int long flags，
const char *device);
```

irq: 是要申请的中断。

handle：中断处理函数指针。

flags：SA_INTERRUPT 请求一个快速中断，0 正常中断。

device：设备名。

如果登记成功，返回 0，这时在/proc/interrupts 文件中可以看你请求的中断。

一些常见的问题

对硬件操作，有时时序很重要。但是如果用 C 语言写一些低级的硬件操作的话，gcc 往往会对你的程序进行优化，这样时序就错掉了。如果用汇编写呢，gcc 同样会对汇编代码进行优化，除非你用 volatile 关键词修饰。最保险的办法是禁止优化。这当然只能对一部分你自己编写的代码。如果对所有的代码都不优化，你会发现驱动程序根本无法装载。这是因为在编译驱动程序时要用到 gcc 的一些扩展特性，而这些扩展特性必须在加了优化选项之后才能体现出来。

第十八章 LINCX 网络设备驱动程序的编写

Linux 系统的设备分为字符设备(char device)，块设备(block device)和网络设备(network device)三种。字符设备是指存取时没有缓存的设备。块设备的读写都有缓存来支持，并且块设备必须能够随机存取(random access)，字符设备则没有这个要求。典型的字符设备包括鼠标，键盘，串行口等。块设备主要包括硬盘软盘设备，CD-ROM 等。一个文件系统要安装进入操作系统必须在块设备上。

网络设备在 Linux 里做专门的处理。Linux 的网络系统主要是基于 BSD unix 的 socket 机制。在系统和驱动程序之间定义有专门的数据结构(sk_buff)进行数据的传递。系统里支持对发送资料和接收资料的缓存，提供流量控制机制，提供对多协议的支持。

一、下面简单介绍一下网络设备驱动程序的一些基本要求。

发送和接收

这是一个网络设备最基本的功能。一块网卡所做的无非就是收发工作。所以驱动程序里要告诉系统你的发送函数在哪里，系统在有资料要发送时就会调用你的发送程序。还有驱动程序由于是直接操纵硬件的，所以网络硬件有资料收到最先能得到这个资料的也就是驱动程序，它负责把这些原始资料进行必要的处理然后送给系统。这里，操作系统必须要提供两个机制，一个是找到驱动程序的发送函数，一个是驱动程序把收到的资料送给系统。是驱动程序把收到的资料送给系统。

中断

中断在现代计算机结构中有重要的地位。操作系统必须提供驱动程序响应中断的能力。一般是把一个中断处理程序注册到系统中去。操作系统在硬件中断发生后调用驱动程序的处理程

时钟

在实现驱动程序时，很多地方会用到时钟。如某些协议里的超时处理，没有中断机制的硬件的轮询等。操作系统应为驱动程序提供定时机制。一般是在预定的时间过了以后回调注册的时钟函数。在网络驱动程序中，如果硬件没有中断功能，定时器可以提供轮询(poll)方式对硬件进行存取。或者是实现某些协议时需要的超时重传等。

二、下面简单介绍一下网络设备驱动程序的需要用的 LINUX 内核函数。

内存申请和释放

include/linux/kernel.h 里声明了 kmalloc()和 kfree()。用于在内核模式下申请和释放内存。

```
void *kmalloc(unsigned int len,int priority);
void kfree(void *__ptr);
```

与用户模式下的 malloc()不同，kmalloc()申请空间有大小限制。长度是 2 的整次方。可以申请的最大长度也有限制。另外 kmalloc()有 priority 参数，通常使用时可以为 GFP_KERNEL，如果在中断里调用用 GFP_ATOMIC 参数，因为使用 GFP_KERNEL 则调用者可能进入 sleep 状态，在处理中断时是不允许的。

kfree()释放的内存必须是 kmalloc()申请的。如果知道内存的大小，也可以用 kfree_s()

request_irq()、free_irq()

这是驱动程序申请中断和释放中断的调用。在 include/linux/sched.h 里声明。

申请中断的调用

request_irq()调用的定义：

```
int request_irq(unsigned int irq,
            void (*handler)(int irq, void *dev_id, struct pt_regs *regs),
            unsigned long irqflags,
            const char * devname,
```

```
void *dev_id);
```

irq 是要申请的硬件中断号。在 Intel 平台，范围 0--15。handler 是向系统登记的中断处理函数。这是一个回调函数，中断发生时，系统调用这个函数，传入的参数包括硬件中断号，device id，寄存器值。dev_id 就是下面的 request_irq 时传递给系统的参数 dev_id。irqflags 是中断处理的一些属性。比较重要的有 SA_INTERRUPT，标明中断处理程序是快速处理程序(设置 SA_INTERRUPT)还是慢速处理程序(不设置 SA_INTERRUPT)。快速处理程序被调用时屏蔽所有中断。慢速处理程序不屏蔽。还有一个 SA_SHIRQ 属性，设置了以后运行多个设备共享中断。dev_id 在中断共享时会用到。一般设置为这个设备的 device 结构本身或者 NULL。中断处理程序可以用 dev_id 找到相应的控制这个中断的设备，或者用 rq2dev_map 找到中断对应的设备。

释放中断的调用

```
void free_irq(unsigned int irq,void *dev_id);
```

时钟

时钟的处理类似中断，也是登记一个时间处理函数，在预定的时间过后，系统时钟的处理类似中断，也是登记一个时间处理函数，在预定的时间过后，系统会调用这个函数。在 include/linux/timer.h 里声明。

```
struct timer_list {
      struct timer_list *next;
      struct timer_list *prev;
      unsigned long expires;
      unsigned long data;
      void (*function)(unsigned long);
};
void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
void init_timer(struct timer_list * timer);
```

使用时钟，先声明一个 timer_list 结构，调用 init_timer 对它进行初始化。time_list 结构里 expires 是标明这个时钟的周期，单位采用 jiffies 的单位。jiffies 是 Linux 一个全局变量，代表时间。它的单位随硬件平台的不同而不同。系统里定义了一个常数 HZ，代表每秒种最小时间间隔的数目。这样 jiffies 的单位就是 1/HZ。Intel 平台 jiffies 的单位是 1/100 秒，这就是系统所能分辨的最小时间间隔了。所以 expires/HZ 就是以秒为单位的这个时钟的周期。

function 就是时间到了以后的回调函数，它的参数就是 timer_list 中的 data。data 这个参数在初始化时钟的时候赋值，一般赋给它设备的 device 结构指针。

在预置时间到系统调用 function,同时系统把这个 time_list 从定时队列里清除。所 I/O

埠的存取使用

```
inline unsigned int inb(unsigned short port);
inline unsigned int inb_p(unsigned short port);
inline void outb(char value, unsigned short port);
inline void outb_p(char value, unsigned short port);
```

在 include/adm/io.h 里定义。

inb_p()、outb_p()与 inb()、outb_p()的不同在于前者在存取 I/O 时有等待

(pause)一适应慢速的 I/O 设备。
为了防止存取 I/O 时发生冲突，Linux 提供对埠使用情况的控制。在使用埠之前，可以检查需要的 I/O 是否正在被使用，如果没有，则把端口标记为正在使用，使用完后再释放。系统提供以下几个函数做这些工作。

```
int check_region(unsigned int from, unsigned int extent);
void request_region(unsigned int from, unsigned int extent,const char *name);
void release_region(unsigned int from, unsigned int extent);
```

其中的参数 from 表示用到的 I/O 端口的起始地址，extent 标明从 from 开始的埠数目。name 为设备名称。

```
void release_region(unsigned int from, unsigned int extent);
```

其中的参数 from 表示用到的 I/O 端口的起始地址，extent 标明从 from 开始的埠数目。name 为设备名称。

中断打开关闭

系统提供给驱动程序开放和关闭响应中断的能力。是在 include/asm/system.h 中的两个定义。

```
#define cli() __asm__ __volatile__ ("cli"::)
```

打印信息

类似普通程序里的 printf()，驱动程序要输出信息使用 printk()。在 include/linux/kernel.h 里声明。

```
int printk(const char* fmt, ...);
```

其中 fmt 是格式化字符串。...是参数。都是和 printf()格式一样的。

注册驱动程序

如果使用模块(module)方式加载驱动程序，需要在模块初始化时把设备注册
到系统设备表里去。不再使用时，把设备从系统中卸除。定义在 drivers/net/net_init.h 里的两个函数完成这个工作。

```
int register_netdev(struct device *dev);
void unregister_netdev(struct device *dev);
```

dev 就是要注册进系统的设备结构指针。在 register_netdev()时，dev 就是要注册进系统的设备结构指针。在 register_netdev()时，dev 结构一般填写前面 11 项，即到 init，后面的暂时可以不用初始化。最重要的是 name 指针和 init 方法。name 指针空(NULL)或者内容为''或者 name[0]为空格(space)，则系统把你的设备做为以太网设备处理。以太网设备有统一的命名格式，ethX。对以太网这么特别对待大概和 Linux 的历史有关。

init 方法一定要提供，register_netdev()会调用这个方法让你对硬件检测和设置。register_netdev()返回 0 表示成功，非 0 不成功。

sk_buff

inux 网络各层之间的资料传送都是通过 sk_buff。sk_buff 提供一套管理缓冲区的方法，是 Linux 系统网络高效运行的关键。每个 sk_buff 包括一些控制方法和一块资料缓冲区。控制方法按功能分为两种类型。一种是控制整个 buffer 链的方法，另一种是控制资料缓冲区的方法。sk_buff 组织成双向链表的形式，根据网络应用的特点，对链表的操作主要是删除链表头的元素和添加到链表尾。sk_buff 的控制方法都很短小以尽量减少系统负荷。(translated from article written by AlanCox)

.alloc_skb() 申请一个 sk_buff 并对它初始化。返回就是申请到的 sk_buff。

.dev_alloc_skb()类似 alloc_skb，在申请好缓冲区后，保留 16 字节的帧头空间。主要用在 Ethernet 驱动程序。

.kfree_skb() 释放一个 sk_buff。

.skb_clone() 复制一个 sk_buff，但不复制资料部分。

.skb_copy()完全复制一个 sk_buff。

.skb_dequeue() 从一个 sk_buff 链表里取出第一个元素。返回取出的 sk_buff

.skb_dequeue() 从一个 sk_buff 链表里取出第一个元素。返回取出的 sk_buff，如果链表空则返回 NULL。这是常用的一个操作。

.skb_queue_head() 在一个 sk_buff 链表头放入一个元素。

.skb_queue_tail() 在一个 sk_buff 链表尾放入一个元素。这也是常用的一个操作。网络资料的处理主要是对一个先进先出队列的管理，skb_queue_tail()和 skb_dequeue()完成这个工作。

.skb_insert() 在链表的某个元素前插入一个元素。

.skb_append() 在链表的某个元素后插入一个元素。一些协议(如 TCP)对没按顺序到达的资料进行重组时用到 skb_insert()和 skb_append()。

.skb_reserve() 在一个申请好的 sk_buff 的缓冲区里保留一块空间。这个空间一般是用做下一层协议的头空间的。

.skb_put() 在一个申请好的 sk_buff 的缓冲区里为资料保留一块空间。在

alloc_skb 以后，申请到的 sk_buff 的缓冲区都是处于空(free)状态，有一个 tail 指针指向 free 空间，实际上开始时 tail 就指向缓冲区头。skb_reserve()在 free 空间里申请协议头空间，skb_put()申请资料空间。见下面的图。

.skb_push() 把 sk_buff 缓冲区里资料空间往前移。即把 Head room 中的空间移一部分到 Data area。

.skb_pull() 把 sk_buff 缓冲区里 Data area 中的空间移一部分到 Head room 中。

```
 --------------------------------------------------
|            Tail room(free)                       |
 --------------------------------------------------
 --------------------------------------------------
            After alloc_skb()

 --------------------------------------------------
| Head room |        Tail room(free)               |
 --------------------------------------------------
            After skb_reserve()

 --------------------------------------------------
| Head room |     Data area      | Tail room(free) |
 --------------------------------------------------
            After skb_put()

 --------------------------------------------------
|Head| skb_ |   Data             | Tail room(free) |
|room| push |                    |                 |
```

```
|    |        Data area          |                  |
 --------------------------------------------------

            After skb_push()


 --------------------------------------------------
|   Head     | skb_ |  Data area | Tail room(free)  |
|            | pull |            |                  |
|            | pull |            |                  |
|    Head room      |            |                  |
 -------------------------------------------------
```

最重要的是网络设备的数据结构。定义在 include/linux/netdevice.h 里。它的注释已经足够详尽。

```
struct device
{

  /*
   * This is the first field of the "visible" part of this structure
   * (i.e. as seen by users in the "Space.c" file).  It is the name
   * the interface.
   */
  char                    *name;

  /* I/O specific fields - FIXME: Merge these and struct ifmap into one */
  unsigned long           rmem_end;             /* shmem "recv" end     */
  unsigned long           rmem_end;             /* shmem "recv" end     */
  unsigned long           rmem_start;           /* shmem "recv" start   */
  unsigned long           mem_end;              /* shared mem end       */
  unsigned long           mem_start;            /* shared mem start     */
  unsigned long           base_addr;            /* device I/O address   */
  unsigned char           irq;                  /* device IRQ number    */

  /* Low-level status flags. */
  volatile unsigned char  start,                /* start an operation   */
                          interrupt;            /* interrupt arrived    */
  /* 在处理中断时 interrupt 设为 1，处理完清 0。 */
  unsigned long           tbusy;                /* transmitter busy must be lon
g
 for

  struct device           *next;

  /* The device initialization function. Called only once. */
  /* 指向驱动程序的初始化方法。 */
  int                     (*init)(struct device *dev);
```

```
/* Some hardware also needs these fields, but they are not part of the
   usual set specified in Space.c. */
/* 一些硬件可以在一块板上支持多个接口，可能用到 if_port。 */
/* 一些硬件可以在一块板上支持多个接口，可能用到 if_port。 */
unsigned char           if_port;              /* Selectable AUI, TP,..*/
unsigned char           dma;                  /* DMA channel          */

struct enet_statistics* (*get_stats)(struct device *dev);

/*
 * This marks the end of the "visible" part of the structure. All
 * fields hereafter are internal to the system, and may change at
 * will (read: may be cleaned up at will).
 */

/* These may be needed for future network-power-down code. */
/* trans_start 记录最后一次成功发送的时间。可以用来确定硬件是否工作正常。*/
unsigned long           trans_start;  /* Time (in jiffies) of last Tx */
unsigned long           last_rx;      /* Time of last Rx              */

/* flags 里面有很多内容，定义在 include/linux/if.h 里。*/
unsigned short          flags;        /* interface flags (a la BSD)   */
unsigned short          family;       /* address family ID (AF_INET)  */
unsigned short          metric;       /* routing metric (not used)    */
unsigned short          mtu;          /* interface MTU value          */

/* type 标明物理硬件的类型。主要说明硬件是否需要 arp。定义在
   include/linux/if_arp.h 里。 */
unsigned short          type;         /* interface hardware type      */

/* 上层协议层根据 hard_header_len 在发送资料缓冲区前面预留硬件帧头空间。*/
unsigned short          hard_header_len;      /* hardware hdr length  */

/* priv 指向驱动程序自己定义的一些参数。*/
void                    *priv;        /* pointer to private data      */

/* Interface address info. */
unsigned char           broadcast[MAX_ADDR_LEN];      /* hw bcast add */
unsigned char           pad;                          /* make dev_addr aligne
d
 to 8
bytes */
unsigned char           dev_addr[MAX_ADDR_LEN];       /* hw address   */
```

```
  unsigned char           addr_len;     /* hardware address length      */
  unsigned long           pa_addr;      /* protocol address             */
  unsigned long           pa_brdaddr;   /* protocol broadcast addr      */
  unsigned long           pa_dstaddr;   /* protocol P-P other side addr */
  unsigned long           pa_mask;      /* protocol netmask             */

  struct dev_mc_list     *mc_list;      /* Multicast mac addresses      */
  int                    mc_count;      /* Number of installed mcasts   */

  struct ip_mc_list      *ip_mc_list;   /* IP multicast filter chain    */
  __u32                 tx_queue_len;   /* Max frames per queue allowed */

  /* For load balancing driver pair support */

  unsigned long            pkt_queue;   /* Packets queued */
  struct device           *slave;       /* Slave device */
  struct net_alias_info         *alias_info;    /* main dev alias info */
  struct net_alias              *my_alias;      /* alias devs */

  /* Pointer to the interface buffers. */
  struct sk_buff_head     buffs[DEV_NUMBUFFS];

  /* Pointers to interface service routines. */
  int                     (*open)(struct device *dev);
  int                     (*hard_start_xmit) (struct sk_buff *skb,
                                    struct device *dev);
  int                     (*hard_header) (struct sk_buff *skb,
                                struct device *dev,
                                unsigned short type,
                                void *daddr,
                                void *saddr,
                                unsigned len);
  int                     (*rebuild_header)(void *eth, struct device *dev,
                                unsigned long raddr, struct sk_buff *skb);
#define HAVE_MULTICAST
  void                    (*set_multicast_list)(struct device *dev);
#define HAVE_SET_MAC_ADDR
  int                     (*set_mac_address)(struct device *dev, void *addr);
#define HAVE_PRIVATE_IOCTL
  int                     (*do_ioctl)(struct device *dev, struct ifreq *ifr, in
t
 cmd);
#define HAVE_SET_CONFIG
  int                     (*set_config)(struct device *dev, struct ifmap *map);
```

```
#define HAVE_HEADER_CACHE
  void                    (*header_cache_bind)(struct hh_cache **hhp, struct de
v
ice
*dev, unsigned short htype, __u32 daddr);
*dev, unsigned short htype, __u32 daddr};
  void                    (*header_cache_update)(struct hh_cache *hh, struct de
v
ice
*dev, unsigned char *  haddr);
#define HAVE_CHANGE_MTU

  struct iw_statistics*   (*get_wireless_stats)(struct device *dev);
};
```

三、CS8900 的工作原理

CS8900 与单片机按照 8 位元方式连接，网卡芯片复位后默认工作方式为 I/O 连接，基址是 300H，下面对它的几个主要工作寄存器进行介绍（寄存器后括号内的数字为寄存器地址相对基址 300H 的偏移量）。

- LINECTL（0112H）
  LINECTL 决定 CS8900 的基本配置和物理接口。在本系统中，设置初始值为 00d3H，选择物理接口为 10BASE-T，并使能设备的发送和接收控制位元。
- RXCTL（0104H）
  RXCTL 控制 CS8900 接收特定资料报。设置 RXTCL 的初始值为 0d05H，接收网络上的广播或者目标位址同本地物理位址相同的正确资料报。
- RXCFG（0102H）
  RXCFG 控制 CS8900 接收到特定资料报后会引发接收中断。RXCFG 可设置为 0103H，这样当收到一个正确的资料报后，CS8900 会产生一个接收中断。
- BUSCT（0116H）
  BUSCT 可控制芯片的 I/O 接口的一些操作。设置初始值为 8017H，打开 CS8900 的中断总控制位元。
- ISQ（0120H）
  ISQ 是网卡芯片的中断状态寄存器，内部映像接收中断状态寄存器和发送中断状态寄存器的内容。
- PORT0（0000H）
  发送和接收资料时，CPU 通过 PORT0 传递资料。
- TXCMD（0004H）
  发送控制寄存器，如果写入资料 00C0H，那么网卡芯片在全部资料写入后开始发送资料。
- TXLENG（0006H）
  发送资料长度寄存器，发送资料时，首先写入发送资料长度，然后将资料通过 PORT0 写入芯片。

以上为几个最主要的工作寄存器（为 16 位），CS8900 支持 8 位元模式，当读或写 16 位资料时，低位字节对应偶位址，高位位元组对应奇地址。例如，向 TXCMD 中写入 00C0H，则

可将 00h 写入 305H，将 C0H 写入 304H。

系统工作时，应首先对网卡芯片进行初始化，即写寄存器 LINECTL、RXCTL、RCCFG、BUSCT。发资料时，写控制寄存器 TXCMD，并将发送资料长度写入 TXLENG，然后将资料依次写入 PORT0 口，如将第一个字节写入 300H，第二个字节写入 301H，第三个字节写入 300H，依此类推。网卡芯片将资料组织为链路层类型并添加填充位和 CRC 校验送到网络同样，单片机查询 ISO 的资料，当有资料来到后，读取接收到的资料帧。读数据时，单片机依次读地址 300H，301H，300H，301H…。

第十九章 WinCE.NET 平台的建立

WinCE.NET 是微软公司专门为嵌入式市场设计，为快速建立下一代智能移动和小内存占用的设备提供的一个健壮的实时操作系统。它很好的延续了桌面 Windows 操作系统接口友好，操作简单等特点，在图形用户接口，多媒体支持方面占有明显的优势，目前已在 PDA 市场，手持设备，工业控制，医疗设备等领域得到了广泛应用。

WinCE.NET 包含大量的新增特性和改进特性，如蓝牙（bluetooth）和 802.11 零配置设

定等无线技术;设备仿真特性使你可以对完整的设备环境进行仿真而无需任何额外的硬件投资；在平台向导方面，使您可以从众多的预置设备设计中进行选择，以便跳跃式的开始你的开发流程；此外，还有丰富的多媒体和 Web 浏览功能，如 Microsoft Internet Explorer6.0 和 Windows Media™编译码器（Codec）和控件。强大的联网能力、强劲的实时性和小内存体积占用以及丰富的多媒体和Web浏览功能使得WindowsCE.NET成为各个不同领域嵌入式操作系统的首选。

从行业应用方面来看，WINCE.NET 操作系统是一个适合下一代互连工业自动化设备的理想小体积嵌入平台。由于使用了 MSMQ（Microsoft Message Queuing）这样的先进应用服务，Windows CE 使实现与工厂生产现场现有 IT 设施的全面集成成为可能。它还具有极大增强了的实时支持以提供时间关键的嵌入应用程序所需要的边界限制、确定性的响应时间和控制。因为 Windows CE 能从闪存磁盘中启动，也就避免了暴露在灰尘、高温、和震动环境下，从而使它可以适应甚至是最恶劣的生产环境。

从开发工具方面来看，微软公司为 WinCE.NET 开发提供了强大的工具支持。在平台开发方面，微软公司提供了功能强大的集成开发环境 PlatformBuilder。用户可以通过该工具完成所有 WINCE 的平台开发方面的工作。在应用软件开发方面，微软公司提供了 Embedded VC++、Visual Studio.NET 等多种工具，用户可以根据自己的需要进行选择。

随着嵌入式领域硬件的高速发展，WinCE.NET 在各个行业得到的应用越来越多，已成为嵌入式操作系统市场的主流之一，而凭借着微软公司的强大研发实力，WinCE.NET 自身也正在飞速地发展。2004 年 7 月，微软公司已最新发布了 WinCE.NET5.0 版本。目前市场上使用的最多的是 2003 年 5 月发布的 WinCE.NET4.2 版本。

第二十章 Embeded VC++4.0 示例实验

在 Microsoft Windows CE .NET 应用程序开发人员目前可以使用的三种开发选择（即 Win32、MFC 和 Microsoft .NET 框架压缩版）当中， MFC 仍然是开发人员最常用的选择。而用于 Windows CE 的基于 MFC 的应用程序都是并且只能使用 eMbedded Visual C++ 4.0 创建。因此 Embeded VC++4.0 也成为 WinCE.NET 应用程序开发领域最常用的开发工具。

EVC++4.0 良好地继承了 Microsoft 另一产品 Visual C++ 的接口风格和开发方式，使

得熟悉 VC++ 的开发人员几乎不需要学习，即可转移到 Embeded VC++ 开发领域。
而 EVC++4.0 的程序架构也和 VC++ 基本相似，这使得许多 PC 机上编写的 VC++ 程序，只需要经过少量的修改即可转移到 Embeded VC++4.0 上编译运行，这极大地节省了开发时间和经费。

EVC++4.0 的开发是基于 SDK 来实现的。 SDK (Software Development Kit) 是一个由 Platform Builder 导出的软件开发包。它是一个包含着定制的平台的所有库，头文件和帮助文件的子集。在 EVC++4.0 开发环境下，应用程序工程必须基于 SDK 才能编译链接。同一个应用程序工程可以在不同的 SDK 下编译链接。如果应用程序中使用了用户自定义的资源，那么该应用程序只有在由用户定制的平台生成的 SDK 下才能正确地编译链接，也只有在用户定制的平台上才能正确运行，这也是为什么许多 WINCE 应用程序转换了平台以后就不能运行的原因之一。